




# Universal Algebra in UniMath

Gianluca Amato   

Università di Chieti–Pescara, Italy

Marco Maggesi   

Università di Firenze, Italy

Cosimo Perini Brogi   

Università di Genova, Italy

---

## Abstract

We present recent updates in our development of a library for Universal Algebra in the UniMath proof assistant. The code here discussed concerns multi-sorted signatures and their algebras, along with the basics for equation systems. Moreover, we give neat constructions of the corresponding univalent categories by using the formalism of displayed categories, and show that the term algebra over a signature is the initial object of the category.

Besides the formalization, we reflect on the methodological principles – based on the idea of evaluability of our elementary construction by the built-in normalization procedure of the system – leading our coding style, and show that this path is practicable indeed by sketching simple examples.

**2012 ACM Subject Classification** Theory of computation → Algebraic language theory; Theory of computation → Type theory; Theory of computation → Automated reasoning; Theory of computation → Constructive mathematics

**Keywords and phrases** Universal Algebra, Homotopy Type Theory, Univalent Foundations, UniMath

**Supplementary Material** Computer formalization in the UniMath system.

*Version 2021-02-02:* <https://github.com/amato-gianluca/UniMath/releases/tag/ITP2021>

**Funding** The authors thank the *European research network on types for programming and verification* (COST Action EUTypes CA15123) for supporting the School and Workshop on Univalent Mathematics 2019 where this project had its origin.

*Gianluca Amato:* Partially supported by INdAM-GNCS and COST Action EUTypes CA15123.

*Marco Maggesi:* Partially supported by the Italian Ministry of University and Research, INdAM-GNSAGA, and COST Action EUTypes CA15123.

## Introduction

In this paper we report on our ongoing implementation of Universal Algebra within the formal environment of UniMath [15].

Our leading motivation has been to provide a general framework for formalizing and studying algebraic structures as presented in the field of Universal Algebra *within* a proof assistant. By providing a formal system for isolating the invariants of the theory we are interested in, Univalent Mathematics has seemed to provide a suitable environment to carry out our endeavour since the very beginning. In particular, since it is natural to study mathematical structures up to isomorphisms, Univalent Mathematics seems to be especially suited for this kind of task.

We chose to work within the UniMath environment since it provides a minimalist implementation of univalent type theory, and, at the same time, it comes with a large repository of mechanized results covering several fields of mathematics, so that it opens a wide range of possibilities for future development of our formalization.

By the code we survey on in the present paper, we have introduced the main notions concerning multi-sorted signatures. Developing them in a formal environment has required

some expedients in the definitions of the basics, and, accordingly, of some subsequent constructions too. In particular, we had to introduce heterogeneous vectors and generalise types involving signatures by introducing (what we called) “sorted sets”.

Having signatures, we then have given the related formalization of the category of algebras using the notion of a displayed category [6] over the category of *sorted* **hSets**, whose univalence is proven by adapting the strategy used for the univalence of functor categories. The resulting construction is still a modular one and the resulting proof-term is more concise, for sure, than the one obtained by checking that algebras and homomorphisms satisfy the axioms for standard categories.

Defining terms is made complex by the fact that, by precise choice, UniMath does not make use of the theory of (Co)Inductive Constructions. Therefore, we encode terms as lists of operation symbols, to be thought of as instructions for a stack-based machine. Terms are those lists of symbols that may be virtually executed without generating type errors or stack underflows.

Moreover, we prove that the term algebra over a signature is the initial object in the corresponding category, and that, more generally, an algebra of terms over a signature and a set of variables has the desired universal mapping property.

Our formalization also includes the notion of equations and of algebras modelling an equation system associated with a signature; as for the category of algebras, we use the displayed category formalism to construct the univalent category of equational algebras over a given signature  $\sigma$  as the full subcategory of algebras over  $\sigma$  satisfying an equation system.

A preliminary version of this work – with single-sorted signatures – has been presented at the Workshop on Homotopy Type Theory and Univalent Mathematics 2020 [9].

## Outline

In the main body of the present work, we will discuss all the notions we introduced in our implementation, so that the paper is structured as a presentation of the code, followed by the discussion of some examples, a quick anticipation of future work and a brief comparison with different formalizations of similar topics as coda. In details, we can summarise it as follows:

- In § 1, we give a general account of the methodology and the principles which guide our work;
- In § 2.1, we introduce the very basics of universal algebra along with some auxiliary definitions involving the new types we need to handle **multi-sorted signatures**, their **algebras**, and **homomorphisms**;
- In § 2.2, we present the main details of our implementation of **terms**, prove that **term algebras** and **free algebras** do have the required universal property – stated as the contractibility of the type of out-going homomorphisms – and discuss the practical and methodological relevance of our **induction principle** on terms;
- In § 2.3, we introduce **systems of equations** and **equational algebras** over a signature;
- In § 2.4, we sketch the main lines of our constructions of the **categories of algebras and equational algebras** over a signature in terms of displayed categories over a base category of **indexed hSets** whose univalence is proven by a proof-strategy very close to that one adopted for functor categories;
- Finally, § 3 is devoted to three applications of our implementation, namely: lists (§ 3.1), monoids (§ 3.2), and Tarski’s semantics of propositional boolean formulas (§ 3.3).

## 1 Goals and methodology

What we have mechanized is not, clearly, a mathematical novelty, but our endeavour has some payoffs.

Firstly, the code introduces in the UniMath library a minimal set of definitions and results that is open to the community of developers for future achievements and formal investigations on the relation between pre-categorical research in general algebraic structures and its subsequent development in e.g. Lawvere theories [1].

Secondly, a peculiar feature of our code is the original implementation of term algebras over a signature, which fits within the original approach to mechanization of univalent mathematics, but highlights the computational relevance of the constructions avoiding the restrictions imposed on the Coq engine in UniMath.

As a matter of fact, terms for a signature constitute a family inductively defined over the operation symbols together with an additional set of variables. As it is known, however, UniMath makes use of just *some* features of the Coq proof assistant. To be precise, both `record` and `inductive` types are avoided in order to keep the system sound from a foundational/philosophical viewpoint.

On the other hand, one of our main goals has been to make *all* our constructions about terms *evaluable* – as far as possible – by the built-in automation mechanisms of the proof assistant. More precisely, we represent each term using a sequence of function symbols. This sequence is thought to be executed by a stack machine: each symbol of arity  $n$  pops  $n$  elements from the stack and pushes a new element at the top. A term is denoted by a sequence of function symbols that a stack-like machine can execute without type errors and stack underflow, returning a stack with a single element.

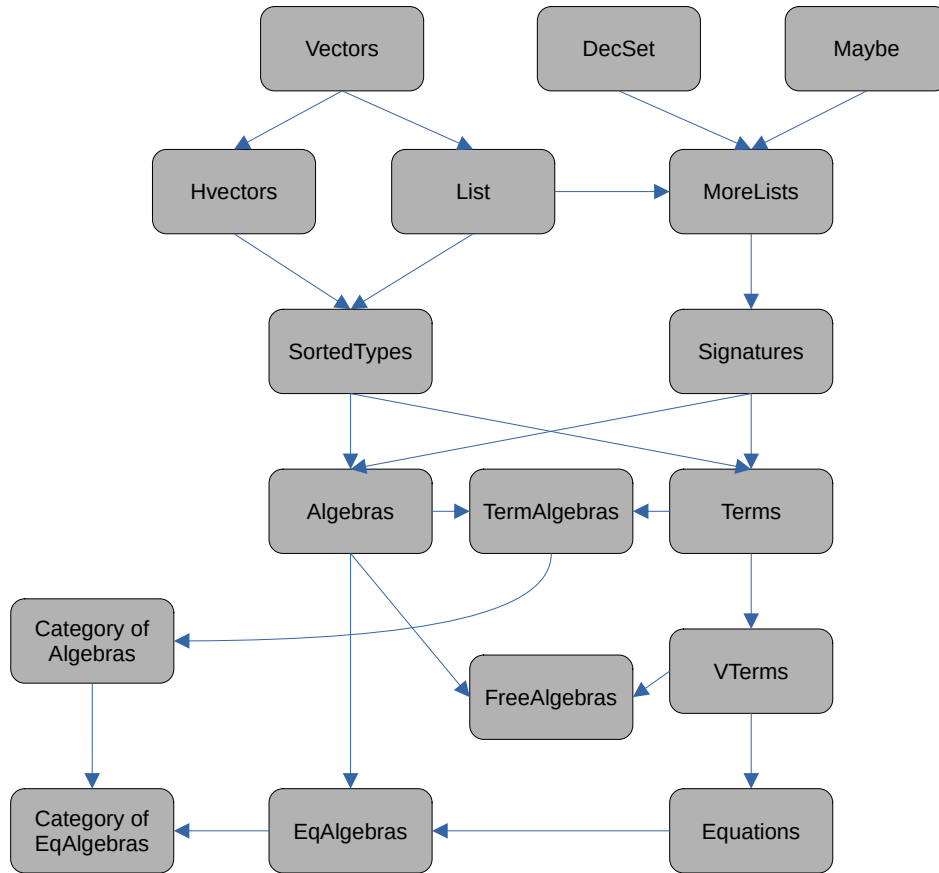
This approach led us to prove both a recursion and induction principle on terms that is evaluable as a functional term of the formal system. This is somehow mandatory when sticking to a form of (small scale) reflection: with our formalized stack-machine we have written in UniMath an implicit algorithm to compute terms over a signature; by means of our induction principle we can run it – so to speak – *within* the very formal system of UniMath, and use it to reason about terms in a safe way.

Moreover, our methodology sympathizes (in a sense) with what Barendregt called Poincaré principle: our implementation of terms allows us to rely on the very core engine of UniMath when dealing with these formal objects, so that whenever we want to handle them we can focus on the real demonstrative contents of the formalization, leaving to the automation behind the computer proof assistant the trivial computational steps involved in the very proof-term.

Generally speaking, we find standard categorical presentations, though perspicaciously elegant in their abstractness, lacking a certain suitability for computerized mathematics. By contrast, our goal is justified by a specific need of methodological coherence – we just sketched it few line above – when approaching a work in formalization. Having proof-terms that the computational machinery of UniMath practically evaluates as a correctly typed function seems to fit the philosophy and aims of the mechanization of mathematics better than just giving a formal counterpart of traditional mathematical notions that the computer cannot handle feasibly.

### Formalization

Definitions and results we discuss in the present work are labelled with their corresponding proof-term identifiers in the formalization files. To improve readability, in what follows, most



■ **Figure 1** Intermodule dependencies of the Universal Algebra formalization in UniMath.

proofs and technicalities are omitted, still they are available in our repository.

Our code is freely accessible from <https://github.com/amato-gianluca/UniMath> and it has already been submitted for inclusion in the official UniMath repository. The revision discussed in this paper is tagged as ITP2021. Our implementation consists of the files in the directories

- `UniMath/Algebra/Universal_Algebra` for the basics of Universal Algebra (together with auxiliary definitions and results), and
- `UniMath/CategoryTheory/categories/Universal_Algebra` for the categories of algebras and equational algebras over a signature.

In order to help readers to browse our library, we summarise the dependencies between the files by the diagram in Figure 1 – where an arrow pointing to a node indicate the dependency of the target from the source.

## 2 The Universal Algebra library

In the following subsections we present and comment on the main constructions constituting our library. As anticipated, in the present work we want to make the code contained in the formalization files easily readable, so that we privilege a certain clarity of exposition over its

exhaustiveness. The interested reader can fill in the details by browsing the related files we mention during the discussion.

## 2.1 Signatures and algebras

We start by defining a **multi-sorted signature** to be made of a *decidable set* of sorts along with operations classified by arities and result sorts, as in standard practice

```
Definition signature : UU :=  $\sum$  (S: decSet) (O: hSet), O  $\rightarrow$  list S  $\times$  S.
```

The three natural projections associated to signatures are named `sort`, `names`, and `ar`

```
Definition sorts ( $\sigma$ : signature) := pr1  $\sigma$ .
Definition names ( $\sigma$ : signature) := pr12  $\sigma$ .
Definition ar ( $\sigma$ : signature) := pr22  $\sigma$ .
```

We also add specific projections for arities

```
Definition sort { $\sigma$ : signature} (nm: names  $\sigma$ ) : sorts  $\sigma$  := pr2 (ar  $\sigma$  nm).
Definition arity { $\sigma$ : signature} (nm: names  $\sigma$ ) : list (sorts  $\sigma$ ) := pr1 (ar  $\sigma$  nm).
```

These simple definitions rely on two files in `UniMath/Combinatorics`, namely `Vectors.v` and `Lists.v`. We change these libraries in two aspects. First of all, we redefine lists in terms of the new datatype `vec` instead of using the ad-hoc type `iterprod` in the standard version of the file. Moreover, we change a couple of theorems from opaque (`Qed. conclusion`) to transparent (`Defined. conclusion`). The latter changes are needed to make terms *compute* correctly.

Note that, in a signature, the set of sorts should be a `decSet`: this is a type whose equality is decidable, as defined in the file `DecSet.v`. We need this extra property because – as we previously stated – we want to *evaluate* terms in the UniMath engine: we can achieve that by pushing sorts into a stack, and we need to check that the very stack contains certain sequences of sorts before applying an operator symbol. Note that a `decSet` also enjoys the defining property of an `hSet`. Operators are only required to be in `hSet`.

A signature may be alternatively specified through the type `signature_simple`. In a simple signature, the types for sorts and operation symbols are standard finite sets, and the map from operations symbols to domain and range is replaced by a list. In this way, the definition of a new signature is made simpler.

```
Definition signature_simple : UU :=  $\sum$  (ns: nat), list (list ([ ns ])  $\times$  [ ns ]).
```

```
Definition make_signature_simple {ns: nat} (ar: list (list ([ ns ])  $\times$  [ ns ]))
  : signature_simple := ns ,, ar.
```

```
Coercion signature_simple_compile ( $\sigma$ : signature_simple) : signature
  := make_signature ([ pr1  $\sigma$  ] ,, isdeceqstn _)
  (stnset (length (pr2  $\sigma$ ))) (nth (pr2  $\sigma$ )).
```

*Single-sorted signatures* are then defined as special cases of `signature_simple`.

```
Definition signature_simple_single_sorted : UU := list nat.
```

```
Definition make_signature_simple_single_sorted (ar: list nat) :
  signature_simple_single_sorted := ar.
```

## 6 Universal Algebra in UniMath

```

Coercion signature_simple_single_sorted_compile
  ( $\sigma$ : signature_simple_single_sorted)
  : signature
  := make_signature_single_sorted (stnset (length  $\sigma$ )) (nth  $\sigma$ ).

```

Moving to the file `Algebras.v`, we define an **algebra** over a given signature  $\sigma$  to be, as usual, support sets indexed by sorts together with operations with appropriate sorts:

```

Definition algebra ( $\sigma$ : signature): UU
  :=  $\sum$  A: shSet (sorts  $\sigma$ ),  $\prod$  nm: names  $\sigma$ , A $\star$  (arity nm)  $\rightarrow$  A (sort nm).

Definition supportset { $\sigma$ : signature} (A: algebra  $\sigma$ ) := pr1 A.

Definition support { $\sigma$ : signature} (A: algebra  $\sigma$ ): sUU (sorts  $\sigma$ ) := pr1 A.

Definition ops { $\sigma$ : signature} (A: algebra  $\sigma$ ) := pr2 A.

Definition dom { $\sigma$ : signature} (A: algebra  $\sigma$ ) (nm: names  $\sigma$ ): UU := A $\star$  (arity nm).

Definition rng { $\sigma$ : signature} (A: algebra  $\sigma$ ) (nm: names  $\sigma$ ): UU
  := support A (sort nm).

```

We declare the projections `supportset`, `support`, and `ops` as type coercions. Moreover, as for signatures, we simplify the building term for algebras when starting from a simple signature:

```

Definition make_algebra_simple
  ( $\sigma$ : signature_simple) (A: Vector hSet (pr1  $\sigma$ ))
  (ops: ( $\lambda$  a, (e1 A) $\star$  (dirprod_pr1 a)  $\rightarrow$  e1 A (dirprod_pr2 a)) $\star$  (pr2  $\sigma$ ))
  : algebra  $\sigma$ .

```

A similar proof-term (`make_algebra_simple_single_sorted`) is given for single-sorted signatures. As you can see, these definitions rely on many different notions and notations. These are introduced in `MoreLists.v`, `SortedTypes.v` and `HVectors.v` files.

The file `MoreLists.v` contains notations for lists, such as `[v1; ...; vn]` for list literals and `::` for *cons*, together with additional properties which cannot be found in the standard library.

The type `hvec` in `HVectors.v` denotes heterogeneous vectors:<sup>1</sup> if  $v$  is a vector of types  $U_1, U_2, \dots, U_n$ , then `hvec v` is the product type  $U_1 \times (U_2 \times \dots \times (U_n \times \text{unit}))$ . We introduce several basic operations on heterogeneous vectors: often they have the same syntax as the corresponding operations on plain vectors, and a name which begins with the prefix `h`. We also introduce notations for heterogeneous vectors, such as `[(v1; ...; vn)]` for a literal and `:::` for prefixing.

Sorted types are types indexed by elements of another type (the index type), so that an element of `sUU S` is an  $S$ -sorted type, i.e. an  $S$ -indexed family of types. Similarly, the type `shSet S` is an  $S$ -indexed family of `hSets`. For functions,  $X \mathbf{s} \rightarrow Y$  denotes the type of  $S$ -sorted mapping between  $X$  and  $Y$ , i.e. of  $S$ -indexed families of functions  $X \mathbf{s} \rightarrow Y \mathbf{s}$ . More prominently, for any  $S$ -sorted type  $X$ , its lifting to `list S` is denoted by  $X\star$ , and is ruled by the identity  $X [s_1; s_2; \dots; s_n] = [X s_1; X s_2; \dots; X s_n]$ . Accordingly, if  $f$  is an indexed mapping between  $S$ -indexed types  $X$  and  $Y$ , then  $f\star\star$  is the lifting of  $f$  to a

---

<sup>1</sup> We need this type to handle operations taking inputs of different sorts.

list  $S$ -indexed mapping between  $X^\star$  and  $Y^\star$ . This operation  $\star\star$  is indeed functorial, and we prove that in a form which does not require function extensionality, since resorting to axioms would break computability of terms.

All of these notions allow us to define **algebra homomorphism**:

```
Definition ishom {σ: signature} {A1 A2: algebra σ} (h: A1 s→ A2) : UU
:= ∏ (nm: names σ) (x: dom A1 nm), h _ (A1 nm x) = A2 nm (h★★ _ x).
```

```
Definition hom {σ: signature} (A1 A2: algebra σ): UU := ∑ (h: A1 s→ A2), ishom h.
```

As expected, the property of being an homomorphism belongs to  $\mathbf{hProp}$ , so that the type  $A1 \simeq A2$  of homomorphisms between  $A1$  and  $A2$  is an  $\mathbf{hSet}$ :

```
Theorem isapropishom {σ: signature} {A1 A2: algebra σ} (f: sfun A1 A2)
: isaprop (ishom f).
```

```
Theorem isasethom {σ: signature} (A1 A2: algebra σ): isaset (A1 ≃ A2).
```

Next, we prove – by lemmas `ishomid` and `ishomcomp` – that the identity function determines an identity homomorphism, and that the property `ishom` is closed under composition.

## 2.2 Terms and free algebras

The file `Algebras.v` is closed by the construction of the unit algebra as the final algebra among those defined over a given signature:

```
Definition unitalgebra (σ: signature): algebra σ
:= make_algebra (sunitset (sorts σ)) tosunit.
```

```
Theorem isconthomstounit {σ: signature} (A: algebra σ)
: iscontr (hom A (unitalgebra σ)).
```

However, we are mostly interested in the initial object of the category of algebras, namely the algebra of terms over a given signature. In standard textbooks, the set of terms over a signature  $\sigma$  and a (disjoint) set  $V$  of variables is defined as the least set including  $V$  and closed under application of symbols of  $\sigma$ .

For being inductive types unavailable in our formal system, we have developed a *peculiar device* to implement that notion. In our formalization we start with the special case where the set of variables  $V$  is empty. The rough and general idea can be sketched as follows:

1. A sequence of function symbols is thought of as a series of commands to be executed by a *stack machine* whose stack is made of sorts, and which we define by means of a maybe monad we construct from `raw` in `Monad.v`:

```
Local Definition oplist (σ: signature):= list (names σ).
```

```
Local Definition stack (σ: signature): UU := maybe (list (sorts σ)).
```

2. When an operation symbol is executed, its arity is popped out from the stack and replaced by its range. When a stack underflow occurs, or when the sorts present in the stack are not the ones expected by the operator, the stack goes into an error condition which is propagated by successive operations. We implement this process by means of two functions: `opexec` and `oplistexec`:

Local Definition `opexec (nm: names  $\sigma$ ): stack  $\sigma \rightarrow$  stack  $\sigma$`   
`:= flatmap ( $\lambda$  ss, just (sort nm :: ss))  $\circ$`   
`flatmap ( $\lambda$  ss, prefix_remove (arity nm) ss).`

Local Definition `oplistexec (l: oplist  $\sigma$ ): stack  $\sigma :=$  foldr opexec (just []) l.`

The former is the stack transformation corresponding to the execution of the operation symbol `nm`. The latter returns the stack corresponding to the execution of the entire `oplist l` starting from the empty stack. The list is executed from the last to the first operation symbol.

Several additional lemmas are required in order to make us able to handle stacks – by concatenating, splitting, etc. – without incurring failures breaking down the whole process, as defined in `Terms.v`.<sup>2</sup>

3. Finally, we define a term to be just a list of operation symbols that, after being executed by `oplistexec`, returns a list of length one with appropriate sort:<sup>3</sup>

Local Definition `isaterm (s: sorts  $\sigma$ ) (l: oplist  $\sigma$ ): UU`  
`:= oplistexec l = just ([s]).`

Local Definition `term ( $\sigma$ : signature) (s: sorts  $\sigma$ ): UU`  
`:=  $\sum$  t: oplist  $\sigma$ , isaterm s t.`

Terms may be built using the `build_term` constructor and decomposed through the `princop` and `subterms` accessors:

Local Definition `build_term (nm: names  $\sigma$ ) (v: (term  $\sigma$ ) $\star$  (arity nm)):`  
`term  $\sigma$  (sort nm).`  
 Definition `princop {s: sorts  $\sigma$ } (t: term  $\sigma$  s): names  $\sigma$ .`  
 Definition `subterms {s: sorts  $\sigma$ } (t: term  $\sigma$  s): (term  $\sigma$ ) $\star$ .`

The implementation function `build_term` is quite straightforward: it concatenates `nm` and the oplists underlying the terms in `v`, and builds a proof that the resulting oplist is a term from the proofs that the elements of `v` are terms. The `princop` and `subterms` accessors are projections of a more complex operation called `term_decompose` which breaks a term in principal operation symbols `nm` and subterms `v`, and, at the same time, provides the proof-terms that characterize their behaviour.

### 2.2.1 Induction on terms

At this point, we proceed in proving induction over terms. The inductive hypothesis, being quite complex, is stated in the `term_ind_HP` type.

Definition `term_ind_HP (P:  $\prod$  (s: sorts  $\sigma$ ), term  $\sigma$  s  $\rightarrow$  UU) : UU`  
`:=  $\prod$  (nm: names  $\sigma$ ) (v: (term  $\sigma$ ) $\star$  (arity nm)) (IH: hvec (hmap_vector P v))`  
`, P (sort nm) (build_term nm v).`

<sup>2</sup> In particular, since we need to decide when a stack is correctly executed and when an underflow occurs, we see the reasons for choosing sorts to constitute a decidable set.

<sup>3</sup> From a purely HoTT-perspective, we can easily see also that the type of stacks over  $\sigma$  is an `hSet`, so that the property of being a term is not proof-relevant (`isapropisaterm`).



Given a family  $P$  of types, indexed by a sort  $s$  and a term over  $s$ , the inductive hypothesis is a function that, given an operation symbol  $nm$ , a sequence of terms  $v$ , and a sequence of proofs of  $P$  for all terms in  $v$ , is able to build a proof of  $P$  for the term `build_term nm v`, i.e.,  $nm(v_1, \dots, v_n)$ . The identifier `h1map_vector` simply denotes the variant of `map` for heterogeneous vectors. Given this auxiliary definition, the induction principle for terms may be easily stated as follows:

```
Theorem term_ind (P:  $\prod$  (s: sorts  $\sigma$ ), term  $\sigma$  s  $\rightarrow$  UU) (R: term_ind_HP P)
  {s: sorts  $\sigma$ } (t: term  $\sigma$  s)
  : P s t.
```

The proof proceeds by induction on the length of the oplist underlying  $t$ , using the `term_ind_onlength` auxiliary function.

Simple examples of use of the induction principle on terms are the `depth` and `fromterm` functions. The former computes the depth of a term, and the latter is essentially the evaluation map for ground terms in an algebra.

```
Local Definition fromterm {A: sUU (sorts  $\sigma$ )}
  (op :  $\prod$  (nm : names  $\sigma$ ), A $\star$  (arity nm)  $\rightarrow$  A (sort nm))
  {s: sorts  $\sigma$ }
  : term  $\sigma$  s  $\rightarrow$  A s
  := term_ind ( $\lambda$  s _, A s) ( $\lambda$  nm v rec, op nm (h2lower rec)).
```

The `h2lower` proof-term which appears in the definition of `fromterm` is just a technicality needed to convert between types which are provably equal but not convertible. This might be replaced by a `transport`, if we were not interested in computability. The same can be said for the proof term `h1lift`, later in the definition of `term_ind`.

In order to reason effectively on inductive definition, we need an induction unfolding property. For natural numbers, it is `nat_rect P a IH (S n) = IH n (nat_rect P a IH n)`, which means that the result of applying the recursive definition to `S n` may be obtained by applying the recursive definition to `n` and then the inductive hypothesis. While this induction unfolding properties are provable just by `reflexivity` for many inductive types, this does not hold for terms, and a quite complex proof is needed:

```
Lemma term_ind_step (P:  $\prod$  (s: sorts  $\sigma$ ), term  $\sigma$  s  $\rightarrow$  UU) (R: term_ind_HP P)
  (nm: names  $\sigma$ ) (v: (term  $\sigma$ ) $\star$  (arity nm))
  : term_ind P R (build_term nm v)
  = R nm v (h2map ( $\lambda$  s t q, term_ind P R t) (h1lift v)).
```

Many of the definition which appears in `Terms.v` are declared as `Local`. This is because they are considered internal implementation details and should not be used if not explicitly needed. In particular, this holds for a set of identifiers that will be redefined in `VTerms.v` to work on terms with variables. Since sometimes it may be convenient to have specialized functions that only work with ground terms, they are exported through a series of notations, such as:

```
Notation gterm := term.
Notation build_gterm := build_term.
```

## 2.2.2 Terms with variables and free algebras

Considering terms with variables is what we do in file `VTerms.v`. The idea is that a term with variables in  $V$  over a signature  $\sigma$  is a ground term in a new signature where constant symbols are enlarged with the variables in  $V$ . Variables and corresponding sorts are declared in a `varspec` (*variable specification*), while `vsignature` builds the new signature.

Definition `varspec` ( $\sigma$ : signature) :=  $\sum V$ : hSet,  $V \rightarrow$  sorts  $\sigma$ .

Definition `vsignature` ( $\sigma$ : signature) ( $V$ : varspec  $\sigma$ ): signature  
:= `make_signature` (sorts  $\sigma$ ) (`setcoprod` (names  $\sigma$ )  $V$ )  
(`sumofmaps` (ar  $\sigma$ ) ( $\lambda v$ , nil, varsort  $v$ )).

The proof-terms `namelift` and `varname` are the injections of, respectively, operation symbols and variables in the extended signature.

Definition `namelift` ( $V$ : varspec  $\sigma$ ) ( $nm$ : names  $\sigma$ ): names (vsignature  $\sigma$   $V$ )  
:= `inl`  $nm$ .

Definition `varname` { $V$ : varspec  $\sigma$ } ( $v$ :  $V$ ): names (vsignature  $\sigma$   $V$ ) := `inr`  $v$ .

Then, a list of definitions comes: they essentially introduce terms with variables by resorting to ground terms.

Definition `term` ( $\sigma$ : signature) ( $V$ : varspec  $\sigma$ )  
: sUU (sorts  $\sigma$ ) := `gterm` (vsignature  $\sigma$   $V$ ).

Definition `build_term` { $V$ : varspec  $\sigma$ } ( $nm$ : names  $\sigma$ ) ( $v$ : (term  $\sigma$   $V$ ) $\star$  (arity  $nm$ ))  
: term  $\sigma$   $V$  (sort  $nm$ ) := `build_gterm` (`namelift`  $V$   $nm$ )  $v$ .

Definition `varterm` { $V$ : varspec  $\sigma$ } ( $v$ :  $V$ )  
: term  $\sigma$   $V$  (varsort  $v$ ) := `build_gterm` (`varname`  $v$ ) [()].

Finally, in `FreeAlgebras.v` we pack terms and the `build_term` operation into the algebra  $T_\sigma(V)$  of terms over a given signature  $\sigma$  and set of variables  $V$ . For this algebra, we prove the universal property.

Definition `free_algebra` ( $\sigma$ : signature) ( $V$ : varspec  $\sigma$ ): algebra  $\sigma$  :=  
`@make_algebra`  $\sigma$  (`termset`  $\sigma$   $V$ ) `build_term`.

Definition `universalmap`  
:  $\sum h$ : free\_algebra  $\sigma$   $V \simeq a$ ,  $\prod v$ :  $V$ ,  $h \_$  (`varterm`  $v$ ) =  $\alpha$   $v$ .

Definition `iscontr_universalmap`  
: `iscontr` ( $\sum h$ : free\_algebra  $\sigma$   $V \simeq a$ ,  $\prod v$ :  $V$ ,  $h$  (varsort  $v$ ) (`varterm`  $v$ ) =  $\alpha$   $v$ ).

In `TermAlgebras.v` we just consider the special case of `FreeAlgebras.v` for the empty set of variables, i.e., for ground terms. In this case, the universal mapping property is replaced by the initiality of the ground term algebra.

### 2.3 Equations and equational algebras

Equations and their associated structures are key notions in Universal Algebra. Although an extensive treatment of notions such as equational algebra and variety is out of the scope of the present work, the basic definitions are already present in our implementation in file `EqAlgebras.v`.

In our setting, an *equation* is a pair of terms (with variables) of the same sort. Their intended meaning is to specify *identities law* where variables are implicitly universally quantified.

Definition `equation` ( $\sigma$ : signature) ( $V$ : varspec  $\sigma$ ): UU  
:=  $\sum s$ : sorts  $\sigma$ , term  $\sigma$   $V$   $s \times$  term  $\sigma$   $V$   $s$ .

The associated projections are denoted `eqsort`, `lhs`, and `rhs` respectively. An *equation system* is just a family of equations.

**Definition** `eqsystem`  $(\sigma : \text{signature}) (V : \text{varspec } \sigma) : \mathbb{U}$   
 $:= \sum E : \mathbb{U}, E \rightarrow \text{equation } \sigma V.$

Then, we pack all the above data into an *equational specification*, that is a signature endowed with an equation system (and the necessary variable specification).

**Definition** `eqspec`  $: \mathbb{U} := \sum (\sigma : \text{signature}) (V : \text{varspec } \sigma), \text{eqsystem } \sigma V.$

The interpretation of an equation is easily defined using function `fromterm` introduced in § 2.2.1. More precisely, the predicate `holds` that checks if the universal closure of an equation `e` holds in an algebra `a` is given as follows:

**Definition** `holds`  $\{\sigma : \text{signature}\} \{V : \text{varspec } \sigma\}$   
 $(a : \text{algebra } \sigma) (e : \text{equation } \sigma V) : \mathbb{U}$   
 $:= \prod \alpha, \text{fromterm } a \alpha (\text{eqsort } e) (\text{lhs } e) = \text{fromterm } a \alpha (\text{eqsort } e) (\text{rhs } e).$

From this, it is immediate to define the type `eqalgebra` of *equational algebras* as those algebras in which all the equations of a given equational specification hold.

## 2.4 Categorical structures

Universal algebra has a natural and fruitful interplay with category theory [12]. As claimed in the introduction, our mechanization includes basic categorical constructions for organizing and reasoning about universal algebra structures. In agreement with the general philosophy of univalent mathematics,<sup>4</sup> we can prove that the categories we are interested in – of algebras and equational algebras – are univalent indeed.

In order to develop formal proofs of that property, two possible strategies are available.

A simplest one consists of building the desired categories from scratch, and then prove that univalence holds between their isomorphic objects. However, experience has shown that this strategy often lacks a certain naturalness, and it makes the steps involved in the construction hard.

The second available strategy has revealed practicable in a more efficient way: we define the desired category in a step-by-step construction by adding *layers* to a *base category* already given. Such a notion of layer corresponds precisely to a *displayed category* [6]: displayed categories are type-theoretic counterpart of fibrations and constitute a widely adopted instrument to reason about categories even at higher dimensions [2] in the UniMath library.

After defining a displayed category over a base category, we can then build a *total category* whose univalence is proven by checking univalence for the base category *and* a displayed version of univalence for the category displayed over the base. This is a generalised version of the so-called *structure identity principle* as introduced first by Aczel as invariance of all structural properties of isomorphic structures (broadly considered).

To build our category of algebras, we apply just this very principle: the structure of algebras and homomorphisms is displayed over a base category of `shSets` that we construct from raw.

At this point, we can easily prove the displayed univalence for that layer, whereas proving that the base category of `shSets` is univalent revealed already non-trivial. We managed on

<sup>4</sup> See the remarks in [5], where category theory was introduced first in a HoTT-setting.

the issue by tweaking the proof-terms already constructed for functor categories in UniMath. The resulting total category of algebras is therefore univalent in the usual sense.

Turning now to equational algebras, we do not have to start the construction again from scratch: within the displayed category formalism we can identify the “substructure” of algebras over `shSets` satisfying a system of equations. In other terms, we can take for equational algebras the layer over the category of `shSets` made of the *full displayed subcategory* of the displayed category of algebras identified by the type `is_eqalgebra`. Again, proving displayed univalence for this layer is not difficult, so that the total category of equational algebras over a system of equations is univalent, as required.

Finally, we rephrase the universal property of the term algebra shown in § 2.2: we can state its initiality in the category of algebras over a given  $\sigma$  by means of the proof-term made of the of the algebra itself and the contractibility of out-going homomorphisms, previously constructed.

### 3 Examples

In this section, we want to illustrate by simple examples how to use our framework in three different settings.

#### 3.1 List algebras

We start with a very simple multi-sorted example, the signature of the list datatype and its algebras. We will show how to specify a signature in our framework and how to interpret a list datatype as an algebra. The code for this example can be found in the module `UniMath.Algebra.Universal.Examples.ListDataType`.

We will need two sorts, one for elements and the other for lists. Correspondingly, we name the two elements  $\bullet 0$  and  $\bullet 1$  of the standard finite set with two elements  $\llbracket 2 \rrbracket$ .

```
Definition elem_sort_idx:  $\llbracket 2 \rrbracket$  :=  $\bullet 0$ .
Definition list_sort_idx:  $\llbracket 2 \rrbracket$  :=  $\bullet 1$ .
```

Our signature for the language of lists will consist of two operation symbols for the usual constructors *nil* and *cons* respectively. Such a signature is encoded with a list of pairs. Each pair describe the input (a list of sorts) and the output (a sort) for the corresponding constructor.

```
Definition list_signature: signature_simple
:= make_signature_simple
  [ ( nil ,, list_sort_idx ) ;
    ( [elem_sort_idx ; list_sort_idx] ,, list_sort_idx ) ]%list.
```

For enhanced readability, we assign explicit names to the operator symbols.

```
Definition nil_idx: names list_signature :=  $\bullet 0$ .
Definition cons_idx: names list_signature :=  $\bullet 1$ .
```

Now, we can endow the list datatype of UniMath (`listset`) with the structure of an algebra over `list_signature` by using the list constructors `nil` and `const`. We fix a type  $A$  for our elements.

```
Variable A : hSet.
```

Then, the class of algebras over `list_signature` is given by

```

Definition list_algebra := make_algebra_simple list_signature
  [( A ; listset A )]
  [( λ _, nil ; λ p, cons (pr1 p) (pr2 p) )].

```

From now on in this section, lemmas are just simple verification of convertibility. They are all proven by reflexivity and the proof scripts are omitted. To begin with, we check that the sort of elements is  $A$  and the sort of lists is given by the associated list datatype:

```

Lemma elem_sort_id : supportset list_algebra elem_sort_idx = A.

```

```

Lemma list_sort_id : supportset list_algebra list_sort_idx = listset A.

```

Next, define the associated algebra constructors. First, let us consider the empty list constructor.

```

Definition list_nil : listset A := ops list_algebra nil_idx tt.

```

As expected, it reduces to the usual *nil* constructor.

```

Lemma list_nil_id : list_nil = @nil A.

```

For the list *cons* constructor, the situation is more complicated. The domain of the constructor is the product  $A \times \text{listset } A \times \text{unit}$ , meaning that the constructor has two (uncurried) arguments

```

Lemma list_cons_dom_id : dom list_algebra cons_idx = A × listset A × unit.

```

Thus, the operation extracted by the *ops* projection has type with the following form

```

Definition list_cons : A × listset A × unit → listset A
  := ops list_algebra cons_idx.

```

That said, our *list\_cons* operation reduces to the usual list *cons*.

```

Lemma list_cons_id (x: A) (l: listset A) : list_cons (x, (l, tt)) = cons x l.

```

### 3.2 Equational algebras of monoids

From now on, we will consider single sorted examples for the sake of simplicity. In this Section, we will discuss the *eqalgebra* of monoids. The code for this example can be found in the module `UniMath.Algebra.Universal.Examples.Monoid`.

To define single sorted signatures, our function `make_signature_simple_single_sorted` is a handy shorthand (previously introduced) that takes a list of natural numbers.

```

Definition monoid_signature := make_signature_simple_single_sorted [2; 0].

```

Monoids are already defined in `UniMath`. Similarly to what we did in the previous section with lists, we endow monoids with the structure of a monoid algebra.

```

Definition monoid_algebra (M: monoid) : algebra monoid_signature
  := make_algebra_simple_single_sorted monoid_signature M
  [( λ p, op (pr1 p) (pr2 p) ;
    λ _, unel M )].

```

Next, we provide a variable specification, i.e. an *hSet* of variables together with a map from variables to sorts. Since `monoid_signature` is single-sorted, the only available sort is `tt`. Then, we build the associated algebra of open terms that will be used to specify the equations of the theory of monoids.

```

Definition monoid_varspec : varspec monoid_signature
  := make_varspec monoid_signature natset (λ _, tt).

Definition Mon : UU := term monoid_signature monoid_varspec tt.
Definition mul : Mon → Mon → Mon := build_term_curried (•0: names monoid_signature).
Definition id : Mon := build_term_curried (•1: names monoid_signature).

```

Term variables are associated to natural numbers. In this case, three variables  $x, y, z$  will suffice for our needs

```

Definition x : Mon := varterm (0: monoid_varspec).
Definition y : Mon := varterm (1: monoid_varspec).
Definition z : Mon := varterm (2: monoid_varspec).

```

Now, we have all the ingredients to specify our equations: the monoid axioms of associativity, left identity, and right identity.

```

Definition monoid_equation : UU := equation monoid_signature monoid_varspec.
Definition monoid_mul_lid : monoid_equation := tt,, make_dirprod (mul id x) x.
Definition monoid_mul_rid : monoid_equation := tt,, make_dirprod (mul x id) x.
Definition monoid_mul_assoc : monoid_equation
  := tt,, make_dirprod (mul (mul x y) z) (mul x (mul y z)).

```

We pack the above equations together into an equation system (`monoid_axioms`) and its associated equational specification (`monoid_eqspec`); finally, we define the class of equational algebras of monoids `monoid_eqalgebra`. We omit the formal construction which is uncomplicated and reduces essentially to uninteresting bookkeeping.

Next, we want to show that every “classical” monoid  $M$  has a natural structure of equational algebra. We have to show that  $M$  is a model for our equation system. Let us begin with the left-identity axiom

```

Lemma holds_monoid_mul_lid : holds (monoid_algebra M) monoid_mul_lid.
Proof.
  intro α. cbn in α.
  change (fromterm (monoid_algebra M) α tt (mul id x) = α 0).
  change (op (unel M) (α 0) = α 0).
  apply lunax.
Qed.

```

As you see, we fix the variable evaluation  $\alpha$ , then we observe that our goal reduces to the same law expressed in the usual language of monoids – `op` for the product, `unel M` for the identity,  $\alpha 0$  for the first variable  $x$  – and then the goal is solved at once by applying the corresponding monoid axiom `lunax`.

The other two laws (for right identity and associativity) are proven in the same way. We can now pack everything into a monoid eqalgebra (definitions `is_eqalgebra_monoid` and `make_monoid_eqalgebra`).

### 3.3 Algebra of booleans and Tarski’s semantics

We now describe a further example based on a simple single sorted algebraic language: the algebra of booleans, and its connectives. The code for this example can be found in the module `UniMath.Algebra.Universal.Examples.Bool`. We consider a language with the usual boolean connectives and constants: truth, falsity, negation, conjunction, disjunction, and implication.

Arities can be specified simply by naturals (the number of arguments). We use the function `make_signature_simple_single_sorted` to build a signature from the list of arities:

```

Definition bool_signature :=
  make_signature_simple_single_sorted [0; 0; 1; 2; 2; 2].

```

Obviously, the type of booleans is already defined in UniMath, together with its usual constants and operations: `false`, `true`, `negb`, `andb`, `orb`, `implb`. Booleans forms an `hSet` which is denoted `boolset`. It is easy to organize these constituents into an algebra for our signature by specifying the translation:

```

Definition bool_algebra :=
  make_algebra_simple_single_sorted bool_signature boolset
  [( λ _, false ;
    λ _, true ;
    λ x, negb (pr1 x) ;
    λ x, andb (pr1 x) (pr12 x) ;
    λ x, orb (pr1 x) (pr12 x) ;
    λ x, implb (pr1 x) (pr12 x) )].

```

Next, we build the algebra of (open) terms, that is, boolean formulae. This is done in two steps. First, we give a variable specification, i.e. a set of type variables:

```

Definition bool_varspec := make_varspec bool_signature natset (λ _, tt).

```

Then, we define the algebra of terms and the associated constructors.

```

Definition T := term bool_signature bool_varspec tt.
Definition bot : T := build_term_curried (•0 : names bool_signature).
Definition top : T := build_term_curried (•1 : names bool_signature).
Definition neg : T → T := build_term_curried (•2 : names bool_signature).
Definition conj : T → T → T := build_term_curried (•3 : names bool_signature).
Definition disj : T → T → T := build_term_curried (•4 : names bool_signature).
Definition impl : T → T → T := build_term_curried (•5 : names bool_signature).

```

Finally, we use the universal property of the term algebra to define the interpretation of boolean formulae:

```

Definition interp (α: assignment bool_algebra bool_varspec) (t: T) : bool :=
  fromterm (ops bool_algebra) α tt t.

```

At this point, we can check the effectiveness of our definitions with some applications. To formulate our tests, we introduce three variables `x`, `y`, `z` – as done in the previous section – and a simple evaluation function `v` for variables that assigns `true` to the variable `x` and `y` (the variable of index 0 and 1) and `false` otherwise.

Now, we can run the interpretation function by using the Coq internal evaluation mechanism (using the vernacular command `Eval strategy in term`). For instance, the evaluation of the formula  $x \wedge (z \rightarrow \neg y)$  becomes:

```

Eval lazy in
  interp (λ n, match n with 0 => true | 1 => true | _ => false end)
    (conj x (impl z (not y))).

```

Note that the choice of the lazy strategy is not accidental. Computations required to evaluate such a proof terms are quite heavy and the standard call by value strategy does not seems able to produce a result in reasonable time.

A few other examples are available in our code as, for instance, a proof of Dummett's tautology:

```

Lemma Dummett :  $\prod$  i, interp i (disj (impl x y) (impl y x)) = true.
Proof.
  intro i. lazy.
  induction (i 0); induction (i 1); apply idpath.
Qed.

```

Notice that this formal proof is just a case analysis for truth-tables in disguise: we instantiate the values of  $x$  and  $y$  by applying `induction` twice, but the remaining job is left to the computing mechanism of Coq, which is able to autonomously verify that the evaluation does yield the value `true` in all cases – we only need to apply `idpath`.

## Conclusions and related works

By the code just surveyed, we covered most of fundamental concepts in Universal Algebra. We plan to enhance our implementation along three directions:

First of all, to streamline the interface provided by the library. At this point, the user of the library is exposed to many technical details only relevant to the implementation, such as: the internal signatures generated by `vsignature` for dealing with variables in terms; the existence of two term algebras, one for ground terms, the other for general terms, while the former should only be a particular case of the latter. We plan to redesign the interface in order to hide the internal details as much as possible. Furthermore, the interface for heterogeneous vectors might be generalized to make the `HVectors` module more useful outside of the scope of our library.

Next, to include more advanced results. On the one side, we plan to complete the treatment of equational algebras by defining the initial algebra of terms modulo equational congruence. On the other side, we want to include some relevant theorems, such as the homomorphism theorems and Birkhoff’s theorem for varieties.

Finally, to extend the library with refined applications and examples of univalent reasoning. This would give evidence that even the minimalist environment of UniMath does allow its user to approach mechanized mathematics with the advantages of both univalent reasoning – to handle equivalent objects as naturally as in informal mathematics – and the automation process of the proof assistant – to be smartly used for performing “internal” implementations in order to leave all computations with no demonstrative significance to the machine.

Different approaches to the field in computerized mathematics are already known.

A classical work on implementing Universal Algebra in dependent type theory is the one of Capretta [10], where he systematically uses setoids in Coq to handle equality on structures. Another attempt, still based on setoids, has been recently carried on in Agda [11]. We share the feeling that Univalent Foundations provide a more principled approach.

Initial semantics furnishes elegant techniques for studying induction and recursion principles in a general setting encompassing applications in programming languages and logic. Assuming univalence, steady research activity produced over the time a number of contributions to the UniMath library, see e.g. [3, 4, 7, 8].

Lynge’s [13] – still under development in [14] – seems to settle in a framework that more closely compares with ours. Despite both our formalization and Lynge’s one assume Univalent Mathematics as formal environment, the study we are proposing here differs from his one by adopting a more foundational perspective. This point of view materialises in our choice of UniMath over CoqHoTT, which is the system adopted by Lynge for his encoding. Moreover, our focus makes the implementation we are proposing different also from categorical treatments mentioned above because of the care we have taken about making the constructions easily evaluable by the very normalization procedure of terms.



---

**References**

---

- 1 Jiří Adámek, Jiří Rosický, and Enrico Maria Vitale. *Algebraic theories: a categorical introduction to general algebra*, volume 184. Cambridge University Press, 2010.
- 2 Benedikt Ahrens, Dan Frumin, Marco Maggesi, and Niels van der Weide. Bicatagories in Univalent Foundations. In Herman Geuvers, editor, *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*, volume 131 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:17, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: <http://drops.dagstuhl.de/opus/volltexte/2019/10512>, doi:10.4230/LIPIcs.FSCD.2019.5.
- 3 Benedikt Ahrens, André Hirschowitz, Ambroise Lafont, and Marco Maggesi. High-Level Signatures and Initial Semantics. In Dan Ghica and Achim Jung, editors, *27th EACSL Annual Conference on Computer Science Logic (CSL 2018)*, volume 119 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:22, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: <http://drops.dagstuhl.de/opus/volltexte/2018/9671>, doi:10.4230/LIPIcs.CSL.2018.4.
- 4 Benedikt Ahrens, André Hirschowitz, Ambroise Lafont, and Marco Maggesi. Modular Specification of Monads Through Higher-Order Presentations. In Herman Geuvers, editor, *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*, volume 131 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:19, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: <http://drops.dagstuhl.de/opus/volltexte/2019/10513>, doi:10.4230/LIPIcs.FSCD.2019.6.
- 5 Benedikt Ahrens, Krzysztof Kapulkin, and Michael Shulman. Univalent categories and the rezk completion. In Maria del Mar González, Paul C. Yang, Nicola Gambino, and Joachim Kock, editors, *Extended Abstracts Fall 2013*, pages 75–76, Cham, 2015. Springer International Publishing.
- 6 Benedikt Ahrens and Peter LeFanu Lumsdaine. Displayed Categories. *Logical Methods in Computer Science*, Volume 15, Issue 1, March 2019. URL: <https://lmcs.episciences.org/5252>, doi:10.23638/LMCS-15(1:20)2019.
- 7 Benedikt Ahrens and Ralph Matthes. Heterogeneous Substitution Systems Revisited. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, volume 69 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:23, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: <http://drops.dagstuhl.de/opus/volltexte/2018/8472>, doi:10.4230/LIPIcs.TYPES.2015.2.
- 8 Benedikt Ahrens and Anders Mörtberg. Some wellfounded trees in unimath. In Gert-Martin Greuel, Thorsten Koch, Peter Paule, and Andrew Sommese, editors, *Mathematical Software – ICMS 2016*, pages 9–17, Cham, 2016. Springer International Publishing.
- 9 Gianluca Amato, Marco Maggesi, Maurizio Parton, and Cosimo Perini Brogi. Universal Algebra in UniMath. In *Workshop on Homotopy Type Theory/Univalent Foundations – HoTT/UF2020*, 2020. URL: <https://hott-uf.github.io/2020/>.
- 10 Venanzio Capretta. Universal algebra in type theory. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs '99*, volume 1690 of *LNCS*, pages 131–148. Springer, 1999. URL: [http://www-sop.inria.fr/lemme/Venanzio.Capretta/universal\\_algebra.html](http://www-sop.inria.fr/lemme/Venanzio.Capretta/universal_algebra.html).
- 11 Emmanuel Gunther, Alejandro Gadea, and Miguel Pagano. Formalization of universal algebra in Agda. *Electronic Notes in Theoretical Computer Science*, 338:147–166, 2018. URL: <https://www.sciencedirect.com/science/article/pii/S1571066118300768>.
- 12 Martin Hyland and John Power. The category theoretic understanding of universal algebra: Lawvere theories and monads. *Electronic Notes in Theoretical Computer Science*, 172:437–458, 2007.

- 13 Andreas Lyngé. Universal algebra in HoTT, 2017. Bachelor's thesis, Department of Mathematics, Aarhus University. URL: <https://github.com/andreaslyn/Work/blob/master/Math-Bachelor.pdf>.
- 14 Andreas Lyngé and Bas Spitters. Universal algebra in HoTT. In *TYPES 2019, 25th International Conference on Types for Proofs and Programs*, 2019. URL: [http://www.ii.uib.no/~bezem/abstracts/TYPES\\_2019\\_paper\\_7](http://www.ii.uib.no/~bezem/abstracts/TYPES_2019_paper_7).
- 15 Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. UniMath — a computer-checked library of univalent mathematics. Available at <https://github.com/UniMath/UniMath>.