Agda Universal Algebra Library

Release 0.2

William DeMeo

Jan 30, 2020

CONTENTS

CHAPTER

ONE

PREFACE

To support formalization in type theory of research level mathematics in universal algebra and related fields, we are developing a software library, called the Agda Universal Algebra Library (aka agda-ualib, aka Agda Algebra). Our library contains formal statements and proofs of some of the core, foundational definitions and results universal algebra.

We will define some useful datatypes for implementing universal algebra in the Agda proof assistant.

1.1 Vision

The idea for the Agda Algebra library originated with the observation that, on the one hand a number of basic and important constructs in universal algebra can be defined recursively, and theorems about them proved inductively, while on the other hand types (of type theory —in particular, dependent types and inductive types) make possible elegant formal representations of recursively defined objects, as well as concise proofs of their properties. These observations suggest that there is much to gain from implementing universal algebra in a language that facilitates working with dependent and inductive types.

Agda is a programming language and proof assistant, or "interactive theorem prover" (ITP), that not only supports dependent and inductive types, but also provides powerful proof tactics[1] for proving properties of the objects that inhabit these types.

The goal of the Agda Algebra project is to formalize, in the Agda language, the substantial edifice upon which our mathematical research stands, demonstrating that our work can be implemented formally and effectively in type theory in such a way that we and other working mathematicians can use the resulting library to conduct and formalize further mathematics research.

Our field is deep and its history rich, so encoding all of our subject's foundations may seem like a daunting task and possibly risky investment of time and resources. However, our view is that the basics of the theory could be well served by a modernized and (where possible) constructive presentation, so that universal algebra could be naturally codified in the language of type theory and formally implemented in, and verified by, the Agda proof assistant.

1.2 Objectives

We wish to emphasize that our ultimate objective is not merely to translate existing results into a more modern and formal language. Indeed, one important goal of the Agda development team is to develop a system that is useful for conducting research in mathematics, and that is how we intend to use our library once we have achieved our immediate objective of implementing the basic foundational core of universal algebra in Agda.

[1] See, e.g., this or that.

To this end, our main objectives include

- developing domain specific "proof tactics" to express the idioms of universal algebra,
- incorporating automated proof search for universal algebra, and
- formalizing theorems emerging from our own mathematics research,
- documenting the resulting software libraries so they are useable by other working mathematicians.

For our own mathematics research, we believe a proof assistant equipped with specialized libraries for universal algebra, as well as domain-specific tactics to automate proof idioms of our field, will be extremely useful. Our goal is to demonstrate (to ourselves and colleagues) the utility of such libraries and tactics for proving new theorems.

1.3 Intended audience

This document describes the Agda Universal Algebra Library (agda-ualib) in enough detail so that working mathematicians (and possibly some normal people, too) might be able to learn enough about Agda and its libraries to put them to use when creating, formalizing, and verifying new mathematics.

While there are no strict prerequisites, we expect anyone with an interest in this work will have been motivated by prior exposure to universal algebra, as presented in, say, [1] or [2], and to a lesser extent category theory, as presented in categorytheory.gitlab.io or [3].

Some prior exposure to type theory and Agda would be helpful, but even without this background one might still be able to get something useful out of this by referring to the appendix and glossary, while simultaneously consulting one or more of the references mentioned in *References* to fill in gaps as needed.

Finally, it is assumed that while reading these materials the reader is actively experimenting with Agda using emacs with its agda2-mode extension installed.

1.4 Installing the library

The main repository for the agda-ualib is https://gitlab.com/ualib/agda-ualib.

There are installation instructions in the main README.md file in that repository, but really all you need to do is have a working Agda (and agda2-mode) installation and clone the agda-ualib repository with, e.g.,

```
git clone git@gitlab.com:ualib/agda-ualib.git
```

OR

```
git clone https://gitlab.com/ualib/agda-ualib.git
```

(We assume you have `Agda` and `agda2-mode` installed on your machine. If not, follow the directions on the main Agda website to install them.)

1.5 Acknowledgments

This manual and the software library that it documents are open access projects maintained on Gitlab. Besides the main authors, a number of other people have contributed to the Agda Algebra project. We are especially grateful to Clifford Bergman, Siva Somayyajula, Venanzio Capretta, Andrej Bauer, Miklós Maróti, Ralph Freese, and Jeremy Avigad for many helpful discussions, as well as the invaluable instruction, advice, and encouragement that they continue to lend to this project (often without knowing it).

1.6 References

The following Agda documentation and tutorials are quite helpful and informed our development.

- Altenkirk, Computer Aided Formal Reasoning
- Bove and Dybjer, Dependent Types at Work
- Escardo, Introduction to Univalent Foundations of Mathematics with Agda
- János, Agda Tutorial
- Norell and Chapman, Dependently Typed Programming in Agda
- Wadler, Programming Language Foundations in Agda

Finally, the official Agda Wiki, Agda User's Manual, Agda Language Reference, and the (open source) Agda Standard Library source code are also quite useful.

CHAPTER

TWO

DATATYPES FOR ALGEBRAS

2.1 Preliminaries

All but the most trivial Agda programs typically begin by importing stuff from existing libraries (e.g., the Agda Standard Library) and setting some options that effect how Agda behaves. In particular, one can specify which logical axioms and deduction rules one wishes to assume.

For example, here's the start of the first Agda source file in our library, which we call `basic.agda`.

```
{-# OPTIONS --without-K --exact-split #-}

--without-K disables Streicher's K axiom
--(see "NOTES on Axiom K" below).

--exact-split makes Agda to only accept definitions
--with the equality sign "=" that behave like so-called
--judgmental or definitional equalities.

open import Level

module basic where

open import Function using (_∘_)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl)
open Eq.≡-Reasoning
open import Relation.Unary

open import Agda.Builtin.Nat public
  renaming ( Nat to ℕ; _-_ to _∸_; zero to nzero; suc to succ )

open import Data.Fin public
  -- (See "NOTE on Fin" section below)
  hiding ( _+_; _<_ )
  renaming ( suc to fsucc; zero to fzero )
```

We don't have the time (or patience!) to describe each of the above directives. Instead, we refer the reader to the above mentioned documentation (as well as the brief *Note on axiom K* below, explaining the `--without-K` option).

2.2 Signatures

We may wish to encode arity as an arbitrary type (which Agda denotes `Set`).

```
record signature₁ : Set₁ where
  field
    ⟨_⟩ₒ : Set         -- operation symbols.
    ⟨_⟩ₐ : ⟨_⟩ₒ -> Set -- Each operation symbol has an arity.
```

If `S : signature₁` is a signature, then `⟨ S ⟩ₒ` denotes the operation symbols of `S`.

If `□ : ⟨ S ⟩ₒ` is an operation symbol, then `⟨ S ⟩ₐ □` is the arity of `□`.

If you don't like denoting operation symbols of `S` by `⟨ S ⟩ₒ`,

then maybe something like this would do better.

```
record signature₂ : Set₁ where
  field
    □ : Set
    ρ : □ -> Set
```

In that case, if `□ : □ S` is an operation symbol, then `(ρ S) □` is the arity of `□`.

However, it may seem more natural to most algebraists for the arity to be a natural number.

So let us define `signature` once and for all as follows:

```
record signature : Set₁ where
  field
    ⟨_⟩ₒ : Set
    ⟨_⟩ₐ : ⟨_⟩ₒ -> ℕ
```

2.3 Operations

```
data operation (γ α : Set) : Set where

ο : ((γ -> α) -> α) -> operation γ α
```

Here, γ is an "arity type" and α is a "universe type".

Example. the `i`-th γ-ary projection operation on α could be implemented like this:

```
π : ∀ {γ α : Set} -> (i : γ) -> operation γ α

π i =  ο λ x -> x i
```

2.4 Algebras

```
open operation
open signature

record algebra' (S : signature) : Set₁ where

  field
    carrier : Set
    ops : (□ : ⟨ S ⟩ₒ)                  --op symbol
      -> (Fin (⟨ S ⟩ₐ □) -> carrier) --tuple of args
          ---------------------------
      ->  carrier
```

If `(A : algebra S)` is an algebra of `signature S`, then `carrier A` would denote the **universe** of `A`.

If `(□ : ⟨ S ⟩ₒ)` is an operation symbol of `S`, then `(op A) □` would denote the **interpretation** of `□` in `A`.

Alternatively…

```
record algebra (S : signature) : Set₁ where

  field
    ⟦_⟧ᵤ : Set
    _⟦_⟧ : (□ : ⟨ S ⟩ₒ)
      ->   (Fin (⟨ S ⟩ₐ □) -> ⟦_⟧ᵤ)
          ---------------------------
      ->   ⟦_⟧ᵤ
```

In that case, if `(A : algebra S)` is an algebra in `signature S`, then `⟦ A ⟧ᵤ` denotes the universe of `A`.

If `□ : ⟨ S ⟩ₒ` is an operation symbol, `A ⟦ □ ⟧` denotes the interpretation of `□` in `A`.

That's a *little* better… but feel free to invent your own syntax!

2.5 Homomorphisms

```
open algebra

record Hom {S : signature}
  (A : algebra S) (B : algebra S) : Set where

  field

    -- The map:
    ⟦_⟧ₕ : ⟦ A ⟧ᵤ -> ⟦ B ⟧ᵤ

    -- The property the map must have to be a hom:
    homo : ∀ {□ : ⟨ S ⟩ₒ}
           (args : Fin (⟨ S ⟩ₐ □) -> ⟦ A ⟧ᵤ)
       ------------------------------------------------
      ->   ⟦_⟧ₕ ((A ⟦ □ ⟧) args) ≡ (B ⟦ □ ⟧) (⟦_⟧ₕ ∘ args)
```

In the next chapter we turn to the important topic of **terms** (the datatypes for which we have defined in the file `free.agda`).

CHAPTER

THREE

DATATYPES FOR TERMS

(The code described in this chapter resides in `free.agda`.)

As usual, we begin by setting some options and importing a few things from the Agda std lib (as well as our definitions from the `basic.agda` file).

```
{-# OPTIONS --without-K --exact-split #-}

open import Level
open import basic
open import algebra
open signature

module free {S : signature}{X : Set} where

open import preliminaries  using (_∪_ ; ∀-extensionality; Σ)
open import Function using (_∘_)
open import Relation.Unary
open import Relation.Binary hiding (Total)
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; sym)
open Eq.≡-Reasoning
import Relation.Binary.EqReasoning as EqR
```

3.1 Terms

We define the inductive family of terms in `signature S` as follows:

```
data Term : Set where

  generator : X -> Term

  node : ∀ (□ : ⟨ S ⟩₀)
    ->    (Fin (⟨ S ⟩₁ □) -> Term)
         -------------------------
    ->     Term
```

3.2 The term algebra

Here is a datatype for the term algebra in `signature S`.

```
open Term

free : algebra S

free = record { ⟦_⟧ᵤ = Term ; _⟦_⟧ = node }
```

3.3 The universal property

We now come to our first proof.

We wish to show that the term algebra is **aboslutely free**.

That is, we must show

1. every `h : X -> ⟦ A ⟧ᵤ` lifts to a hom from `free` to `A`;
2. the induced hom is unique.

Here is the Agda code proving these facts.

1. a. Every map `(X -> A)` "lifts".

```
free-lift : {A : algebra  S}
            (h : X -> ⟦ A ⟧ᵤ)
          ----------------------
  ->          ⟦ free ⟧ᵤ -> ⟦ A ⟧ᵤ

free-lift h (generator x) = h x

free-lift {A} h (node  args) = (A ⟦  ⟧) λ{i -> free-lift {A} h (args i)}
```

   b. The lift is a hom.

```
open Hom

lift-hom : {A : algebra S}
           (h : X -> ⟦ A ⟧ᵤ)
         ------------------
  ->         Hom free A

lift-hom {A} h =
  record
  {
    ⟦_⟧ₕ = free-lift {A} h;
    homo = λ args -> refl
  }
```

2. The lift to `(free -> A)` is unique.

   N.B. we need function extensionality for this, which we import from our util.agda file (see the `agda-ualib` gitlab repository).

```
free-unique : {A : algebra S}
  ->     ( f g : Hom free A )
  ->     ( ∀ x  ->  ⟦ f ⟧ₕ (generator x) ≡ ⟦ g ⟧ₕ (generator x) )
  ->     (t : Term)
         ---------------------------
  ->     ⟦ f ⟧ₕ t ≡ ⟦ g ⟧ₕ t

free-unique {A} f g p (generator x) = p x

free-unique {A} f g p (node 𝒇 args) =
   begin
     ⟦ f ⟧ₕ (node 𝒇 args)
   ≡⟨ homo f args  ⟩
     (A ⟦ 𝒇 ⟧) (λ i -> ⟦ f ⟧ₕ (args i))
   ≡⟨ cong ((A ⟦_⟧)_) (∀-extensionality (induct f g p args)) ⟩
     (A ⟦ 𝒇 ⟧) (λ i -> ⟦ g ⟧ₕ (args i))
   ≡⟨ sym (homo g args) ⟩
     ⟦ g ⟧ₕ (node 𝒇 args)
   ∎
   where
    induct : {A : algebra S}
      ->      (f g : Hom free A)
      ->      (∀ x -> ⟦ f ⟧ₕ (generator x) ≡ ⟦ g ⟧ₕ (generator x))
      ->      (args : Fin (⟨ S ⟩ₐ 𝒇) → Term)
      ->      (i : Fin (⟨ S ⟩ₐ 𝒇))
              ------------------------------------- (IH)
      ->      ⟦ f ⟧ₕ (args i) ≡ ⟦ g ⟧ₕ (args i)
    induct f' g' h' args' i' = free-unique f' g' h' (args' i')
```

Now that we have see where and how induction is used, let's clean up the proof by inserting the induction step within the angle brackets inside the calculational proof.

```
free-unique : {A : algebra S}
  ->     ( f g : Hom free A )
  ->     ( ∀ x  ->  ⟦ f ⟧ₕ (generator x) ≡ ⟦ g ⟧ₕ (generator x) )
  ->     (t : Term)
         ---------------------------
  ->     ⟦ f ⟧ₕ t ≡ ⟦ g ⟧ₕ t

free-unique {A} f g p (generator x) = p x

free-unique {A} f g p (node 𝒇 args) =
   begin
     ⟦ f ⟧ₕ (node 𝒇 args)
   ≡⟨ homo f args  ⟩
     (A ⟦ 𝒇 ⟧) (λ i -> ⟦ f ⟧ₕ (args i))
   ≡⟨ cong ((A ⟦_⟧)_)
      ( ∀-extensionality λ i -> free-unique f g p (args i) ) ⟩
     (A ⟦ 𝒇 ⟧) (λ i -> ⟦ g ⟧ₕ (args i))
   ≡⟨ sym (homo g args) ⟩
     ⟦ g ⟧ₕ (node 𝒇 args)
   ∎
```

Due to time constraints, this is as far as I got during my JMM lecture.

Please check the web site ualib.org for more information.

(As of 23 Jan 2020, the ualib.org website is outdated. We plan to remedy this by early February 2020, and update the site with links to the latest version of the Agda Universal Algebra Library, as well as documentation describing the library in great detail.)

CHAPTER

FOUR

DATATYPES FOR SUBUNIVERSES AND SUBALGEBRAS

(The code described in this chapter resides in `subuniverse.agda`.)

As usual, we begin by setting some options and importing some modules.

```
{-# OPTIONS --without-K --exact-split #-}

open import Level
open import basic
open algebra
open signature

module subuniverse {ℓ : Level} {S : signature} where

open import preliminaries
open import Data.Empty
open import Data.Unit.Base using (⊤)
open import Data.Product
open import Data.Sum using (_⊎_; [_,_])
open import Function
open import Relation.Unary
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl)
open import Data.Product using (Σ; _,_; ∃; Σ-syntax; ∃-syntax)
```

4.1 Subuniverses

To test whether a subset of a universe is a subuniverse, first we have to decide how to model subsets.

One option is to model a subset of $[\![A]\!]_u$ as a *predicate* (i.e., *unary relation*) on $[\![A]\!]_u$.

The `Pred` type is defined in the Agda standard library, in the file `Relation/Unary.agda`, as follows:

```
Pred : ∀ {a}
  ->   Set a  ->    (ℓ : Level)
     --------------------------
  ->      Set (a ⊔ suc ℓ)

Pred A ℓ = A -> Set ℓ
```

So if we let `B : Pred [[ A ]]u ℓ`, then `B` is simply a function of type `A -> Set ℓ`.

If we consider some element `x : [[ A ]]u`, then `x ∈ B` iff `B x` "holds" (i.e., is inhabited).

Next, we define a function `OpClosed` which asserts that a given subset, `B`, of $[\![\ A\ ]\!]_u$ is closed under the basic operations of `A`.

```
OpClosed : (A : algebra S) (B : Pred ⟦ A ⟧u ℓ) -> Set ℓ
OpClosed A B = ∀{𝓸 : ⟨ S ⟩o}
               (args : Fin (⟨ S ⟩a 𝓸) -> ⟦ A ⟧u)
  ->           ( ∀(i : Fin (⟨ S ⟩a 𝓸)) -> (args i) ∈ B )
               ---------------------------------------
  ->           (A ⟦ 𝓸 ⟧) args ∈ B
```

In other terms, `OpClosed A B` asserts that for every operation symbol 𝓸 of `A`, and for all tuples `args` of arguments, if the antecedent `(args i) ∈ B` holds for all `i` (i.e., all arguments belong to `B`), then `(A ⟦ 𝓸 ⟧) args` also belongs to `B`.

Finally, we define the `IsSubuniverse` type as a record with two fields: (1) a subset and (2) a proof that the subset is closed under the basic operations.

```
record IsSubuniverse {A : algebra S} : Set (suc ℓ) where

  field
    sset : Pred ⟦ A ⟧u ℓ       -- a subset of the carrier,
    closed : OpClosed A sset   -- closed under the operations of A
```

To reiterate, we have `sset : Pred ⟦ A ⟧u ℓ`, indicating that `sset` is a subset of the carrier $[\![\ A\ ]\!]_u$, and `closed : OpClosed A sset` indicating that `sset` is closed under the operations of `A`.

4.2 Subalgebras

Finally, we define a datatype for subalgebras of a given algebra `A`. We choose record with three fields:

1. a subset, `subuniv`, of `A`;
2. operations, which are the same as `A` (we could be pedantic and require the operations be restricted to the subset `subuniv`, but this is unnecessary);
3. a proof, named `closed`, that `subuniv` is closed under the operations of `A`.

```
record subalgebra (A : algebra S) : Set (suc ℓ) where

  field

    subuniv : Pred ⟦ A ⟧u ℓ

    _⟦_⟧ : (𝓸 : ⟨ S ⟩o)
      ->   (args : Fin (⟨ S ⟩a 𝓸) -> ⟦ A ⟧u)
      ->   ( ∀(i : Fin (⟨ S ⟩a 𝓸)) -> (args i) ∈ subuniv )
           ---------------------------------------------
      ->   Set ℓ

    closed : OpClosed A subuniv


open IsSubuniverse

SubAlgebra : (A : algebra S)
```

(continues on next page)

(continued from previous page)

```
  ->          (B : IsSubuniverse {A})
             ------------------------
  ->          (subalgebra A)

SubAlgebra A B =
  record
    {
    subuniv = sset B ;
    _⟦_⟧ = λ 𝑓 args p -> (sset B) ((A ⟦ 𝑓 ⟧) args) ;
    closed = closed B
    }
```

CHAPTER

FIVE

APPENDIX

5.1 Note on axiom K

nlab describes **axiom K** as follows[1] :

> "[when added, axiom K turns intensional type theory] into extensional type theory —or more precisely, what is called here 'propositionally extensional type theory.' In the language of homotopy type theory , this means that all types are h-sets , accordingly axiom K is incompatible with the univalence axiom .
>
> "Heuristically, the axiom asserts that each term of each identity type $\texttt{Id}_a\,\texttt{(x,x)}$ (of equivalences of a term `x` of type `a`) is propositionally equal to the canonical reflexivity equality proof $\texttt{refl}_x\ \texttt{:}\ \texttt{Id}_a\,\texttt{(x, x)}$.
>
> "See also extensional type theory – Propositional extensionality ."

5.2 Writing definitions interactively

Here is a description of some Agda key-bindings and how to use them, as we briefly mentioned in the talk.

1. Add a question mark and then type `C-c C-l` to create a new "hole."
2. Type `C-c C-f` to move into the next unfilled hole.
3. Type `C-c C-c` (from inside the hole) to be prompted for what type should fill the given hole.
4. Type `t` (or whatever variable you want to induct on) to split on the variable in the hole.
5. Type `C-c C-f` to move into the next hole.
6. Type `C-c C-,` to get the type required in the current hole.
7. Enter an appropriate object in the hole and type `C-c C-space` to remove the hole.

SUMMARY.

1. `?` then `C-c C-l` creates hole
2. `C-c C-f` moves to next hole
3. `C-c C-c` prompts for what goes in hole
4. `m` splits (inducts) on variable `m`
5. `C-c C-,` in hole gets type required

[1] source: https://ncatlab.org/nlab/show/axiom+K+%28type+theory%29 accessed on: 29 Jan 2020

6. `C-c C-space` removes hole

5.3 REFERENCES

The following is a list of my favorite books on topics that are of fundamental importance to the Agda Universal Algebra Library:

- Backhouse, et al: Algebraic & Coalgebraic Methods in the Maths of Program Construction
- Bergman: Universal Algebra
- Crole: Categories for Types
- McKenzie, McNulty, Taylor: Algebras, lattices, varieties. Vol. I
- Mitchell: Foundations for Programming Languages
- Riehl: Category Theory in Context
- Taylor: Practical Foundations for Mathematics
- The HoTT Book

If you click on one of these links and purchase a book, Amazon will donate a few cents towards the development of this website and the agda-ualib.

BIBLIOGRAPHY

[1] Clifford Bergman. *Universal algebra*. Volume 301 of Pure and Applied Mathematics (Boca Raton). CRC Press, Boca Raton, FL, 2012. ISBN 978-1-4398-5129-6. Fundamentals and selected topics.

[2] Ralph N. McKenzie, George F. McNulty, and Walter F. Taylor. *Algebras, lattices, varieties. Vol. I*. The Wadsworth & Brooks/Cole Mathematics Series. Wadsworth & Brooks/Cole Advanced Books & Software, Monterey, CA, 1987. ISBN 0-534-07651-3.

[3] E. Riehl. *Category Theory in Context*. Aurora: Dover Modern Math Originals. Dover Publications, 2017. ISBN 9780486820804. URL: http://www.math.jhu.edu/~eriehl/context/.