Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 338 (2018) 147–166

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Formalization of Universal Algebra in Agda

Emmanuel Gunther[1] Alejandro Gadea[2] Miguel Pagano[3]

FaMAF, UNC – Córdoba, Argentina

Abstract

In this work we present a novel formalization of universal algebra in Agda. We show that heterogeneous signatures can be elegantly modelled in type-theory using sets indexed by arities to represent operations. We prove elementary results of heterogeneous algebras, including the proof that the term algebra is initial and the proofs of the three isomorphism theorems. We further formalize equational theory and prove soundness and completeness. At the end, we define (derived) signature morphisms, from which we get the contra-variant functor between algebras; moreover, we also proved that, under some restrictions, the translation of a theory induces a contra-variant functor between models.

Keywords: universal algebra, formalization of mathematics, equational logic

1 Introduction

Universal algebra [2] is the study of different types of algebraic structures at an abstract level, thus revealing common results which are valid for all of them and also allowing for a unified definition of constructions (for example, products, sub-algebra, congruences). Universal algebra has played a relevant role in computer science since its earliest days, in particular Birkhoff's seminal paper [4] features regular languages as a prominent example; shortly before, Burstall [6] had proved properties of programs using structural induction, by conceiving the language as an initial algebra. The ADJ group [14] promoted multi-sorted algebras as a key theoretical tool for specifying data types [19], semantics [20], and compilers [33]. More recently, institutions [15], a generalization of universal algebra, have been used as a foundation of methodologies and frameworks for software specification and development [30].

In spite of the rich mathematical theory of heterogeneous algebras (mostly inherited from the monosorted setting, but not always [32]), there are few publicly

[1] Email:gunther@famaf.unc.edu.ar Partially supported by a CONICET scholarship.

[2] Email:gadea@famaf.unc.edu.ar Partially supported by a CONICET scholarship.

[3] Email:pagano@famaf.unc.edu.ar

https://doi.org/10.1016/j.entcs.2018.10.010

available formalizations in type theory (which we discuss in the conclusion). This situation is to be contrasted with impressive advances in mechanization of particular algebraic structures as witnessed, for example, by the proof of the Feit-Thompson theorem in Coq by Gonthier and his team [21].

In this work we present an Agda library of multi-sorted universal algebra aiming both a reader with a background in the area of algebraic specifications and also the community of type theory. For the former, we try to explain enough Agda in order to keep the paper self-contained; we will recall the most important definitions of universal algebra. The main contributions of this paper are: (i) the first formalization of basic universal algebra in Agda; (ii) the first, to our knowledge, formalization in type theory of derived signature morphisms and the reduct algebras induced by them; (iii) a novel representation of heterogeneous signatures in type theory, where operations are modelled using sets indexed by arities; and (iv) an independent library of heterogeneous vectors. We formalized the proof that the term algebra is initial and also the proofs of the three isomorphism theorems; moreover we also define a deduction system for conditional equational logic and prove its soundness and completeness with respect to Goguen and Meseguer's semantics [17]. We also showed that the translations of theories arising from derived signature morphisms induces a contra-variant functor between models. In the complete development, which is available at `https://github.com/manugunther/agda-universal-algebra.git`, we include several examples featuring both the use of equational reasoning and the preservation of models by signature morphisms.

Outline. In Sec. 2 we introduce the basic concepts of Universal Algebra: signature, algebras and homomorphisms, congruences, quotients and subalgebras, the proofs of three isomorphisms theorems, and the proof of the initiality of the term algebra. In Sec. 3 we define an equational calculus, introducing concepts of equations, theories, satisfiability and provability, ending with Birkhoff's proofs of soundness and completeness. In Sec. 4 we introduce a new representation of (derived) signature morphisms and reduct algebras (and homomorphisms), and we explore translation and implication of theories. Finally, we conclude in Sec. 5, discussing the work done, and pointing out possible future directions.

2 Universal Algebra

In this section we present our formalization in Agda of the core concepts of heterogeneous universal algebra; in the next two sections we focus respectively on equational logic and signature morphisms. Meinke' and Tucker's chapter [25] is our reference for heterogeneous universal algebra; we will recall some definitions and state all the results we formalized. Bove et al. [5] offer a gentle introduction to Agda; we expect the reader to be familiar with Haskell or some other functional language.

2.1 Signature, algebra, and homomorphism

Signature

A *signature* is a pair of sets (S, F), called *sorts* and *operations* (or *function symbols*) respectively; each operation is a triple $(f, [s_1, \dots, s_n], s)$ consisting of a *name*, its *arity*, and the *target sort* (we also use the notation $f\colon [s_1, ..., s_n] \Rightarrow s$).

In Agda we use dependent records to represent signatures; in dependent records the type of some field may depend on the value of a previous one or parameters of the record. Type-theoretically one can take operations (of a signature) as a family of sets indexed by the arity and target sort (an indexed family of sets can also be thought as predicates over the index set, an index satisfies the predicate if its family is inhabited):

```
record Signature : Set₁ where
  field
    sorts : Set
    ops   : List sorts × sorts → Set
```

A × B corresponds to the non-dependent cartesian product of A and B.

In order to declare a concrete signature one first declares the set of sorts and the set of operations, which are then bundled together in a record. For example, the mono-sorted signature of monoids has a unique sort, so we use the unit type ⊤ with its sole constructor tt. We define a family indexed on List ⊤ x ⊤, with two constructors, corresponding with the operations: a 0-ary operation e, and a binary operation • (note that constructors can start with a lower-case letter or any symbol):

```
data monoid-op : List ⊤ × ⊤ → Set where
  e : monoid-op ([ ] , tt)
  • : monoid-op ([ tt , tt ] , tt)
monoid-sig : Signature
monoid-sig = record {sorts = ⊤; ops = monoid-op}
```

The signature of monoid actions has two sorts, one for the monoid and the other for the set on which the monoid acts.

```
data actMonₛ : Set where
  mon : actMonₛ
  set : actMonₛ
data actMonₒ : List actMonₛ × actMonₛ → Set where
  e : actMonₒ ([ ] , mon)
  * : actMonₒ ([ mon , mon ] , mon)
  • : actMonₒ ([ mon , set ] , set)
actMon-sig : Signature
actMon-sig = record {sorts = actMonₛ; ops = actMonₒ}
```

Defining operations as a family indexed by arities and target sorts carries some benefits in the use of the library: as in the above examples, the names of operations are constructors of a family of datatypes and so it is possible to perform pattern matching on them. Notice also that infinitary signatures can be represented in our setting; in fact, all the results are valid for any signature, be it finite or infinite.

We show two examples of signatures with infinite operations, the first might be more appealing to computer scientists and the second is more mathematical. The abstract syntax of a language for arithmetic expressions may have one sort, a constant operation for each natural number and a binary operation representing the addition of two expressions.

```
data Sortsₑ : Set where E : Sortsₑ
data Opsₑ : List Sortsₑ × Sortsₑ → Set where
  val  : (n : ℕ) → Opsₑ ([] , E)
  plus : Opsₑ (E :: [ E ] , E)
```

Vector spaces over a field can be seen as a heterogeneous signature with two sorts [4] or as homogeneous signature over the field [3, p. 132]; this latter approach can be easily specified in our library, even if the field is infinite:

```
data Sorts-v Set where V : Sorts-v
data Ops-v (F : Set) : Set where
  _+_ : Ops-v (V :: [ V ] , V)       -- vector addition
  v : (f : F) → Ops-v ([ V ] , V)    -- scalar multiplication
vspace-sig : (F : Set) → Signature
vspace-sig F = record {sorts = Sorts-v; ops = Ops-v F}
```

Algebra

An *algebra* $\mathcal{A}$ for the signature Σ consists of a family of sets indexed by the sorts of Σ and a family of functions indexed by the operations of Σ. We use $\mathcal{A}_s$ for the *interpretation* or the *carrier* of the sort s; given an operation $f\colon [s_1, ..., s_n] \Rightarrow s$, the interpretation of f is a total function $f_{\mathcal{A}}\colon \mathcal{A}_{s_1} \times ... \times \mathcal{A}_{s_n} \to \mathcal{A}_s$. We formalize the product $\mathcal{A}_{s_1} \times ... \times \mathcal{A}_{s_n}$ as *heterogeneous vectors.* The type of heterogeneous vectors is parameterized by a set I and a family of sets indexed by I; and is indexed over a list of I:

```
data HVec {I : Set} (A : I → Set) : List I → Set where
  ⟨⟩    : HVec A []
  _▷_  : ∀ {i is} → A i → HVec A is → HVec A (i :: is)
```

The first parameter I is implicit (written in braces), which means that Agda will infer it by unification; notices that the constructor _▷_ also takes two implicit arguments (we use the notation ∀ to skip their types). Let Σ be a signature and A : sorts $\Sigma \to$ Set, then the product $\mathcal{A}_{s_1} \times ... \times \mathcal{A}_{s_n}$ is formalized as HVec A [s_1,. . . ,s_n].

We need one more ingredient to give the formal notion of algebras: the mathematical definition of carriers assumes an underlying notion of equality. In type theory one makes it apparent by using setoids (i.e. sets paired with an equivalence relation), which were thoroughly studied by Barthe et al. [1]. Setoids are defined in the standard library [9] of Agda[4] as a record with three fields.

```
record Setoid : Set₁ where
  field
    Carrier : Set
    _≈_     : Carrier → Carrier → Set
    isEquivalence : IsEquivalence _≈_
```

The relation is given as a family of types indexed over a pair of elements of the carrier (a b : Carrier are related if the type a ≈ b is inhabited); IsEquivalence _≈_ is again a record whose fields correspond to the proofs of reflexivity, symmetry, and transitivity.

The finest equivalence relation over any set is given by the *propositional equality* which only equates each element with itself, thus we can endow any set with a setoid structure with the function setoid : Set → Setoid of the standard library; vice versa, there is a forgetful functor ‖_‖ : Setoid → Set which returns the carrier.

Setoid morphisms are functions which preserve the equality:

```
record _⟶≈_ (A B : Setoid) : Set where
  field
  _⟨$⟩_ : ‖ A ‖ → ‖ B ‖
  cong : ∀ {a a'} → _≈_ A a a' → _≈_ B (_⟨$⟩ a) (_⟨$⟩ a')
```

Notice that $\_\overset{\approx}{\longrightarrow}\_$ is a record parameterized on two setoids. The first field is the function, by declaring it mixfix one can write f ⟨$⟩ a when f : A $\overset{\approx}{\longrightarrow}$ B and a : ‖ A ‖; the second field is given by a function mapping equivalence proofs on the source setoid to equivalence proofs on the target. Setoid morphisms will be used to give the interpretation of operations.

Let A : I → Set be a family of sets and P : {i : I} → A i → Set a family of predicates, we let P * : ∀ {is} → HVec A is → Set be the point-wise extension of P over heterogeneous vectors. We also use the point-wise extension to define the setoid of heterogeneous vectors given a family of setoids A : I → Setoid and write A ∗ is for the setoid of heterogeneous vectors with index is. Algebras are formalized as records parameterized on the signature.

```
record Algebra (Σ : Signature) : Set₁ where
  field
    _⟦_⟧ₛ : sorts Σ → Setoid
    _⟦_⟧ₒ : ∀ {ar s} → (f : ops Σ (ar , s)) → _⟦_⟧ₛ ∗ ar ⟶≈ _⟦_⟧ₛ s
```

If A is an algebra for the signature monoid-sig, then A ⟦ tt ⟧$_s$ is the carrier, A ⟦ e ⟧$_o$

[4] Our formalization is based on several concepts defined in the standard library.

and A ⟦ • ⟧$_o$ are the interpretations of the operations. We invite the interested reader to browse the examples to see algebras for the signatures we have shown.

Homomorphism

Let Σ be a signature and let $\mathcal{A}$ and $\mathcal{B}$ be algebras for Σ. A *homomorphism* h from $\mathcal{A}$ to $\mathcal{B}$ is a family of functions indexed by the sorts $h_s : \mathcal{A}_s \to \mathcal{B}_s$, such that for each operation $f : [s_1, ..., s_n] \Rightarrow s$, the following holds:

$$h_s(f_{\mathcal{A}}(a_1, ..., a_n)) = f_{\mathcal{B}}(h_{s_1}\, a_1, ..., h_{s_n}\, a_n) \tag{1}$$

Notice that this is a condition over the family of functions.

In order to formalize homomorphisms we first introduce a notation for families of setoid morphisms indexed over sorts:

```
_⇝_ : ∀ {Σ} → Algebra Σ → Algebra Σ → Set
_⇝_ {Σ} A B = (s : sorts Σ) → A ⟦ s ⟧s ─≈→ B ⟦ s ⟧s
```

We make explicit the implicit parameter Σ because otherwise sorts Σ does not make sense.[5] To enforce (1) we also define a predicate over families of setoids morphisms:

```
homCond : ∀ {Σ} {A B} → A ⇝ B → Set
homCond {Σ} {A} {B} h = ∀ {ar s} (f : ops Σ (ar , s)) (as : ‖ A ⟦_⟧s ∗ ar ‖) →
  h s ⟨$⟩ (A ⟦ f ⟧o ⟨$⟩ as) ≈s B ⟦ f ⟧o ⟨$⟩ map h as
```

where _≈$_s$_ is the equivalence relation of the setoid B ⟦ s ⟧$_s$ and map h is the obvious extension of h over vectors. A homomorphism is a record parameterized by the source and target algebras

```
record Homo {Σ} (A B : Algebra Σ) : Set where
  field
    ′_′ : A ⇝ B
    cond : homCond ′_′
```

As expected, we have the identity homomorphism Id$_h$ A : Homo A A and the composition G ∘$_h$ F : Homo A C of homomorphisms F : Homo A B and G : Homo B C. It is also expected that F ∘$_h$ Id$_h$ A and F are equal in some sense. Since Agda is based on an intensional type theory, we cannot take the definitional equality (which distinguishes id from λ n → n + 0 as functions on naturals); instead, we equate setoid morphisms whenever their function parts are extensionally equal:

```
_≈ext_ : (f g : A ─≈→ B) → Set
f ≈ext g = ∀ (a : ‖ A ‖) → (f ⟨$⟩ a) ≈B (g ⟨$⟩ a)
```

Two homomorphisms are equal when their corresponding setoid morphisms are extensionally equal:

[5] In the library we use modules in order to avoid the repetition of the parameters Σ, A, and B.

```
_≈h_ : ∀ {Σ} {A B} → Homo A B → Homo A B → Set
F ≈h F' = (s : sorts Σ) → ′ F ′ s ≈ext ′ F' ′ s
```

With respect to this equality, it is straightforward to prove the associativity of the composition $\_\circ_h\_$ and that Id_h is the identity for the composition.

2.2 *Quotient and subalgebras*

In order to prove the more basic results of universal algebra, we need to formalize subalgebras, congruence relations, and quotients.

Subalgebra

A subalgebra $\mathcal{B}$ of an algebra $\mathcal{A}$ consists of a family of subsets $\mathcal{B}_s \subseteq \mathcal{A}_s$, that are closed under the interpretation of operations; that is, for every $f : [s_1, \ldots, s_n] \Rightarrow s$ the following condition holds

$$(a_1, \ldots, a_n) \in \mathcal{B}_{s_1} \times \cdots \times \mathcal{B}_{s_n} \text{ implies } f_{\mathcal{A}}(a_1, \ldots, a_n) \in \mathcal{B}_s \quad . \tag{2}$$

As shown by Salvesen and Smith [29], subsets cannot be added as a construction in intensional type theory because they lack desirable properties. If A : Set and P : A → Set is a predicate over A, then one can represent the subset containing the elements on A that satisfy P as the dependent sum[6] Σ[a ∈ A] P whose inhabitants are pairs (a , p) where a : A and p : P a. Let us consider a setoid A and a predicate on its carrier P : ∥ A ∥ → Set; first notice that we can lift the subset construction to setoids, defining the equivalence relation (a , q) ≈ (a' , q') iff a ≈ a'. Moreover, we might assume that P is *well-defined*, which means that a $\approx_A$ a' and P a imply P a'.

```
WellDef : (A : Setoid) → (P : ∥ A ∥ → Set) → Set
WellDef A P = ∀ {a a'} → a ≈A a' → P a → P a'
```

A family of well-defined predicates will induce a subalgebra; but we still need to formalize the condition (2). Let Σ be a signature and A be an algebra for Σ.

```
opClosed : (P : (s : sorts Σ) → ∥ A ⟦ s ⟧s∥ → Set) → Set
opClosed P = ∀ {ar s} (f : ops Σ (ar , s)) → (P * ⟨→⟩ P s) (A ⟦ f ⟧o ⟨$⟩_)
```

(Q ⟨→⟩ R) f can be read as the pre-condition Q implies post-condition R after applying f; so opClosed P f asserts that if a vector a* satisfies the predicate P, then the application of the interpretation A ⟦ f ⟧$_o$ to a* satisfies P, according to Eq. (2). In summary, given an algebra A for the signature Σ and a family P of predicates, such that P s is well-defined for every sort s and P is opClosed, we can define the SubAlgebra A P

```
SubAlgebra : ∀ {Σ} A P → WellDef P → opClosed P → Algebra Σ
```

[6] Do not confuse the syntax Σ[_∈_]_ of dependent sum, with a variable Σ : Signature

In the subalgebra, an operation f is interpreted by applying the interpretation of f in A to the first components of the argument (and use the fact that P is op-closed to show that the resulting value satisfies the predicate of the target sort).

Congruence and Quotients

A *congruence* on a Σ-algebra $\mathcal{A}$ is a family Q of equivalence relations indexed by sorts, and each of them is closed under the operations of the algebra. This condition is called *substitutivity* and can be formalized using the point-wise extension of Q over vectors: for every operation $f : [s_1, \ldots, s_n] \Rightarrow s$

$$(\boldsymbol{a}, \boldsymbol{b}) \in Q_{s_1} \times \cdots \times Q_{s_n} \text{ implies } (f_{\mathcal{A}}(\boldsymbol{a}), f_{\mathcal{A}}(\boldsymbol{b})) \in Q_s \tag{3}$$

As with predicates, we say that a binary relation over a setoid is well-defined if it is preserved by the setoid equality; this notion can be extended over families of relations in the obvious way. In our formalization, a congruence on an algebra A is a family Q of well-defined, equivalence relations. The substitutivity condition (3) is aptly captured by the generalized containment operator _=[_]⇒_ of the standard library, where P =[f]⇒ Q if, for all a,b ∈ A, (a,b) ∈ P implies (f a, f b) ∈ Q.

```
record Congruence (A : Algebra Σ) : Set where
  field
    rel : (s : sorts Σ) → (∥ A ⟦ s ⟧ₛ ∥ → ∥ A ⟦ s ⟧ₛ ∥ → Set)
    welldef : (s : sorts Σ) → WellDefBin (rel s)
    cequiv : (s : sorts Σ) → IsEquivalence (rel s)
    csubst : ∀ {ar s} → (f : ops Σ (ar , s)) → rel * =[ A ⟦ f ⟧ₒ ⟨$⟩_ ]⇒ rel s
```

Given a congruence Q over the algebra $\mathcal{A}$, we can obtain a new algebra, the *quotient algebra*, by interpreting the sort s as the set of equivalence classes $\mathcal{A}_s/Q$; the condition (3) ensures that the operation $f : [s_1, \ldots, s_n] \Rightarrow s$ can be interpreted as the function mapping the vector $([a_1], \ldots, [a_n])$ of equivalence classes into the class $[f_{\mathcal{A}}(a_1, \ldots, a_n)]$. In Agda, we take the same carriers from A and use Q s as the equivalence relation over ∥ A ⟦ s ⟧$_s$ ∥; operations are interpreted just as in A and the congruence proof is given by csubst Q.

Isomorphism Theorems

The definitions of subalgebras, quotients, and epimorphisms (surjective homomorphisms) are related by the three isomorphism theorems. Although there is some small overhead by the coding of subalgebras, the proofs follow very close what one would do in paper. For proving these results we also defined the *kernel* and the *homomorphic* image of homomorphisms.

Theorem 2.1 (First isomorphism theorem) *If $h : \mathcal{A} \to \mathcal{B}$ is an epimorphism, then $\mathcal{A}/ker\, h \simeq \mathcal{B}$.*

Remember that the quotient $\mathcal{A}/ker\,h$ has the same carrier as $\mathcal{A}$, so h counts as the underlying function and it respects the equivalence relation $ker\,h$ by definition. Clearly h is surjective and its injectivity is obvious.

Theorem 2.2 (Second isomorphism theorem) *If ϕ, ψ are congruences over $\mathcal{A}$, such that $\psi \subseteq \phi$, then $(\mathcal{A}/\phi) \simeq (\mathcal{A}/\psi)/(\phi/\psi)$.*

In order to prove this theorem, we first prove that ϕ/ψ is a congruence over $\mathcal{A}/\psi$: it suffices to prove the well-definedness of ϕ/ψ, i.e. that $(a, c) \in \psi$, $(b, d) \in \psi$, and $(a, b) \in \phi$ imply $(c, d) \in \phi$; an obvious consequence of $\psi \subseteq \phi$. Notice that the underlying carriers are the same in both cases: those of $\mathcal{A}$, so the identity function is the mediating isomorphism and the proof that it satisfies the homomorphism condition is trivial.

Theorem 2.3 (Third isomorphism theorem) *Let $\mathcal{B}$ be a subalgebra of $\mathcal{A}$ and ϕ be a congruence over $\mathcal{A}$. Let $[\mathcal{B}]^\phi = \{K \in A/\phi : K \cap B \neq \emptyset\}$ and let ϕ_B be the restriction of ϕ to $\mathcal{B}$, then (i) ϕ_B is a congruence over $\mathcal{B}$;(ii) $[\mathcal{B}]^\phi$ is a subalgebra of $\mathcal{A}$; and,(iii) $[\mathcal{B}]^\phi \simeq \mathcal{B}/\phi_B$.*

First we define the *trace* of the congruence ϕ on the subalgebra $\mathcal{B}$ as the restriction of ϕ on $\mathcal{B}$; proving that it is a congruence over $\mathcal{B}$ involves some bureaucracy (remember that an element of a subalgebra is a pair (a, p) such that $a \in A$ and p is the proof that a satisfies the predicate defining B). For the second item, we model $[\mathcal{B}]^\phi$ as a predicate over $\mathcal{A}$; it is satisfied by $a \in A$ if there is some $b \in B$ such that $(a, b) \in \phi$. The well-definedness of this predicate is easy (assuming $(a, a') \in \phi$ and $b \in B$ with $(a, b) \in \phi$, one can easily prove that $(a', b) \in \phi$, thus b is also the witness for proving that a' satisfies the predicate). To prove that the predicate is closed under the operations we take a vector of triples (as, bs, ps) consisting of a vector of elements in A, a vector of elements in B, and the proofs ps proving that $(as_i, bs_i) \in \phi$. Let f be an operation, since B is closed we know $f(b_1, \ldots, b_n) \in B$ and because ϕ is also closed we deduce $(f(a_1, \ldots, a_n), f(b_1, \ldots, b_n)) \in \phi$. Finally, the underlying function witnessing the isomorphism $[\mathcal{B}]^\phi \simeq \mathcal{B}/\phi_B$ is given by composing the second projection with the first projection, thus getting an element in B.

2.3 The Term Algebra is initial

A Σ-algebra $\mathcal{A}$ is called *initial* if for any Σ-algebra $\mathcal{B}$ there exists exactly one homomorphism from $\mathcal{A}$ to $\mathcal{B}$. We give an abstract definition of this universal property, existence of a unique element, for any set A and any relation R

```
hasUnique {A} _≈_ = A × (∀ a a' → a ≈ a')
```

and initiality can be formalized directly:

```
Initial : ∀ {Σ} → Algebra Σ → Set
Initial {Σ} A = ∀ (B : Algebra Σ) → hasUnique (_≈ₕ_ A B)
```

Given a signature Σ we can define the *term algebra* $\mathcal{T}$, whose carriers are sets of well-typed words built up from the function symbols. Sometimes this universe is called the *Herbrand Universe* and is inductively defined:

$$\frac{t_1 \in \mathcal{T}_{s_1} \qquad \cdots \qquad t_n \in \mathcal{T}_{s_n}}{f\,(t_1, ..., t_n) \in \mathcal{T}_s}\; f : [s_1, ..., s_n] \Rightarrow s$$

This inductive definition can be written directly in Agda:

```
data HU {Σ : Signature} : (s : sorts Σ) → Set where
  term : ∀ {ar s} → (f : ops Σ (ar ↦ s)) → HVec HU ar → HU s
```

We use propositional equality to turn each HU_s into a setoid, thus completing the interpretation of sorts. To interpret an operation $f\colon [s_1, \ldots, s_n] \Rightarrow s$ we map the vector $\langle \mathsf{t}_1,\ldots,\mathsf{t}_n\rangle$: HVec HU $[\mathsf{s}_1,\ldots,\mathsf{s}_n]$ to term f $\langle \mathsf{t}_1,\ldots,\mathsf{t}_n\rangle$; we omit the proof of cong, which is too long and tedious to be shown.

```
|T| : (Σ : Signature) → Algebra Σ
|T| Σ = record { _⟦_⟧ₛ = setoid ∘ (HU {Σ}); _⟦_⟧ₒ = |_|ₒ }
  where | f |ₒ = record { _⟨$⟩_ = term f; cong = ... }
```

Terms can be interpreted in any algebra $\mathcal{A}$, yielding an homomorphism $h_A\colon \mathcal{T} \to \mathcal{A}$

$$h_A(f(t_1, \ldots, t_n)) = f_{\mathcal{A}}\,(h_A\, t_1, ..., h_A\, t_n)\ .$$

We cannot translate this definition directly in Agda, instead we have to mutually define |h| and its extension over vectors |h*|

```
|h| : ∀ {Σ} → (A : Algebra Σ) → {s : sorts Σ} → HU s → ∥ A ⟦ s ⟧ₛ ∥
|h| A (term f ts) = A ⟦ f ⟧ₒ ⟨$⟩ (|h*| ts)
```

It is straightforward to prove that |h| preserves propositional equality and satisfies the homomorphism condition by construction. To finish the proof that |T| Σ is initial, we prove, by recursion on the structure of terms, that any pair of homomorphisms are extensionally equal.

3 Equational Logic

In this section we introduce the notion of (conditional) equational theories and the corresponding notion of satisfiability of theories by algebras. Moreover we formalize (conditional) equational logic as presented by Goguen and Lin [16] and prove that the deduction system is sound and complete.

3.1 Free algebra with variables

The term algebra we have just defined contained only *ground* terms, i.e. terms without variables. Given a signature Σ and X : sorts Σ → Set a family of variables, we define a new signature extending Σ with X by taking the variables as new constants (i.e. , operations with arity []).

```
_〈|_|〉 : (Σ : Signature) → (X : sorts Σ → Set) → Signature
Σ 〈| X |〉 = record {sorts = sorts Σ; ops = ops'}
  where ops' ([] , s) = ops Σ ([] , s) ⊎ X s
        ops' (ar , s) = ops Σ (ar , s)
```

Note that it is easy to refer to constant operations and extend them, because we indexed the set of operations on their arity and target sort.

It is easy to turn the term algebra of the extended signature into an algebra for the original signature:

```
|T|_〈|_|〉 : (Σ : Signature) → (X : sorts Σ → Set) → Algebra Σ
|T| Σ 〈| X |〉 = record { _⟦_⟧ₛ = |T| (Σ 〈| X |〉) ⟦_⟧ₛ , _⟦_⟧ₒ = io}
  where io {[]} f = |T| (Σ 〈| X |〉) ⟦ inj₁ f ⟧ₒ
        io {ar} f = |T| (Σ 〈| X |〉) ⟦ f ⟧ₒ
```

The only difference with the algebra of ground terms is that we inject constants from Σ to distinguish them from variables. In order to interpret terms with variables we need *environments* to give meaning to variables.

Let Env X A $= \forall$ {s} $\to$ X s $\to$ $\|$ A $[\![$ s $]\!]_s$ $\|$ be the set of environments from X to A. The free algebra |T| Σ ⦇ X ⦈ has the universal *freeness* property: given A : Algebra Σ and an environment θ : Env X A, there exists an unique homomorphism $[\![\_]\!]\theta$: Homo (|T| Σ ⦇ X ⦈) A such that $[\![$ x $]\!]\theta = \theta$ (x) for x $\in$ X.

3.2 Satisfiability and provability

Equations

In the mono-sorted setting an equation is a pair of terms where all the variables are assumed to be universally quantified and an equational theory is a (finite) set of equations. In a multi-sorted setting both sides of an equation should be terms of the same sort. Moreover we allow quasi-identities which we write as conditional equations:

$$t = t' \text{ if } t_1 = t'_1, \ldots, t_n = t'_n \enspace .$$

Let Σ be a signature and X : sorts $\Sigma \to$ Set be a family of variables for Σ. An identity e : Eq Σ X s is a pair of (open) terms with sort s. A conditional equation is modelled as record with fields for the conclusion and the conditions, modelled as an heterogeneous vector of sorted identities. We declare a constructor to use the lighter notation ⋀ eq if (ar , eqs) instead of **record** {eq = e; cond = (ar , eqs)}.

```
record Equation (Σ : Signature) (X : sorts Σ → Set) (s : sorts Σ) : Set where
  constructor ⋀_if_
  field
    eq   : Eq Σ X s
    cond : Σ[ ar ∈ List (sorts Σ) ] (HVec (Eq Σ X) ar)
```

A *theory* over the signature Σ is given by a vector of conditional equations.

```
Theory : (Σ : Signature) → (X : sorts Σ → Set) → (ar : List (sorts Σ)) → Set
Theory Σ X ar = HVec (Equation Σ X) ar
```

We deviate from Goguen's and Lin's in that we assume that all the equations of a theory share the same set of variables, while they assume that each equation has its own set of quantified variables. Clearly, this simplification is harmless; if we have a theory where each equation has its own set of variables, we can take the union of those sets as the common set. As stressed by Goguen and Meseguer [18], quantifying equations is essential:

[...] the naive unsorted rules of deduction for equational logic (namely, reflexivity, symmetry, transitivity and substitutivity) are not sound when extended to the many-sorted case in the obvious way; [...] adding variable declarations to these rules yields a rule set that is sound.

Satisfiability

Let Σ be a signature and $\mathcal{A}$ be an algebra for Σ. We say that a conditional equation $t = t'$ if $t_1 = t'_1, \ldots, t_n = t'_n$ is *satisfied* by $\mathcal{A}$ if for any environment $\theta : X \to \mathcal{A}$, $[\![t]\!]\theta = [\![t']\!]\theta$, whenever $[\![t_i]\!]\theta = [\![t'_i]\!]\theta$ for $1 \leqslant i \leqslant n$. In order to formalize satisfiability we first define when an environment models an equation.

```
_⊨ₑ_ : ∀ {Σ X A} → (θ : Env X A) → {s : sorts Σ} → Eq Σ X s → Set
_⊨ₑ_ θ {s} (t , t') = _≈_ (A ⟦ s ⟧ₛ) (⟦ t ⟧ θ) (⟦ t' ⟧ θ)
```

Using the point-wise extension of this relation we can write directly the notion of satisfiability.

```
_⊨_ : ∀ {Σ X} (A : Algebra Σ) → {s : sorts Σ} → Equation Σ X s → Set
A ⊨ (⋀ eq if (_ , eqs)) = ∀ θ → ((θ ⊨ₑ_) * eqs) → θ ⊨ₑ eq
```

We say that $\mathcal{A}$ is a *model* of the theory E if it satisfies each equation in E. As usual an equation is a logical consequence of a theory, if every model of the theory satisfies the equation.

```
_⊨ₘ_ : ∀ {Σ X ar} → (A : Algebra Σ) → (E : Theory Σ X ar) → Set
A ⊨ₘ E = (A ⊨_) * E
_ ⊨Σ _ : ∀ {Σ X ar s} → (E : Theory Σ X ar) → (e : Equation Σ X s) → Set
_ ⊨Σ _ {Σ} E e = (A : Algebra Σ) → A ⊨ₘ E → A ⊨ e
```

Provability

As noticed by Huet and Oppen [22], the definition of a sound deduction system for multi-sorted equality logic is more subtle than expected. We formalize the system presented in [16], shown in Fig. 1. The first three rules are reflexivity, symmetry and

$$\frac{}{E \vdash \forall X,\, t = t} \qquad \frac{E \vdash \forall X,\, t_0 = t_1}{E \vdash \forall X,\, t_1 = t_0} \qquad \frac{E \vdash \forall X,\, t_0 = t_1 \qquad E \vdash \forall X,\, t_1 = t_2}{E \vdash \forall X,\, t_0 = t_2}$$

$$\frac{\forall Y,\, t = t' \text{ if } t_1 = t'_1, \ldots, t_n = t'_n \in E \qquad E \vdash \forall X,\, \sigma(t_i) = \sigma(t'_i)}{E \vdash \forall X,\, \sigma(t) = \sigma(t')}\ \sigma\colon Y \to T_\Sigma(X)$$

$$\frac{E \vdash \forall X,\, t_1 = t'_1 \qquad \cdots \qquad E \vdash \forall X,\, t_n = t'_n}{E \vdash \forall X,\, f\,(t_1, \ldots, t_n) = f\,(t'_1, \ldots, t'_n)}\ f : [s_1, ..., s_n] \Rightarrow_\Sigma s$$

Fig. 1. Deduction system

transitivity; the fourth rule, called substitution, allows to instantiate an axiom with a substitution σ, provided one has proofs for every condition of the axiom;[7] finally, the last rule internalizes Leibniz rule, for replacing equals by equals in subterms. Notice that we can only prove identities and not quasi-identities. We define the relation of provability as an inductive type, parameterized in the theory E, and indexed by the conclusion of the proof. For conciseness, we only show the constructor for transitivity:

```
data _⊢_ {Σ X ar} (E : Theory Σ X ar) : ∀ {s} → Eq Σ X s → Set where
  ptrans : ∀ {s} {t₀ t₁ t₂} →
                    E ⊢ (t₀ , t₁) → E ⊢ (t₁ , t₂) → E ⊢ (t₀ , t₂)
```

Let E be a theory over a signature Σ. It is straightforward to define a setoid over |T| Σ ⦇ X ⦈ by letting t₁ ≈ t₂ if E ⊢ t₁ ≈ t₂; this equivalence relation (thanks to the first three rules) is a congruence (because of the last rule) over the term algebra. We can also use the facility provided by the standard library to write proofs with several transitive steps more nicely, as can be seen in the next example.

Soundness and completeness are proved as in the mono-sorted case. For soundness one proceeds by induction on the derivations; completeness is a consequence of the fact that the quotient of the term algebra by provable equality is a model.

Theorem 3.1 (Soundness and Completeness) *$E \vdash t \approx t'$ iff $E \models_\Sigma t \approx t'$.*

Let us remark that completeness does not imply that there is a decidability algorithm for every theory; i.e. this result gives no decision procedure at all.

Let E and E' be two theories over the signature Σ. We say that E is *stronger* than E' if every axiom $e \in E'$ can be deduced from E, written $E \vdash_T E'$. Obviously if E is stronger than E', then any equation that can be deduced from E' can also be deduced from E and any model of E is also a model of E'.

3.3 A theory for Boolean Algebras

In this section we outline how to formalize an equational theory and illustrate each step by showing snippets of the formalization of a Boolean Theory presented by Rocha and Meseguer [28].[8]

[7] In our formalization this rule is slightly less general because we assume all the equations are quantified over the same set of variables.

[8] The full code is available in the file `Examples/EqBool.agda` of the repository.

1. *Define the signature* describing the language, and choose a family of sets for the variables. It helps if one also introduce an abbreviation for terms over the signature extended with variables.

```
data bool-ops : List ⊤ × ⊤ → Set where
  f t : bool-ops ([] ↦ tt)
  neg : bool-ops ([ tt ] ↦ tt)
  and or : bool-ops (([ tt , tt ]) ↦ tt)

bool-sig : Signature
bool-sig = record {sorts = ⊤; ops = bool-ops}
vars : sorts bool-sig → Set
vars tt = ℕ

Form : Set
Form = HU bool-sig ⦇ vars ⦈
```

2. *Introduce smart-constructors* for terms of the extended signature with variables to ease writing the axioms and proving theorems. Usually one has a smart-constructor for each operation and one per variable that is used in the axioms or the theorems.

```
true false : Form
true = term (inj₁ t) ⟨⟩
false = term (inj₁ f) ⟨⟩

p q : Form
p = term (inj₂ 0) ⟨⟩
q = term (inj₂ 1) ⟨⟩

_∧_ : Form → Form → Form
φ ∧ ψ = term and ⟨ φ , ψ ⟩

¬ : Form → Form
¬ φ = term neg ⟨ φ ⟩
```

3. *Define the equational theory* by specifying one equation for each axiom and collect them in a theory; here one can appreciate the convenience of the smart-constructors. Here we only show two of the twelve axioms of the theory bool-theory. If one will prove theorems of the theory, then it is also convenient to define pattern-synonyms for the proofs that each axiom is in the theory.

```
commAnd leastDef : Equation bool-sig vars tt
commAnd = ⋀ (p ∧ q) ≈ (q ∧ p) if ([] , ⟨⟩)
leastDef = ⋀ (p ∧ (¬ p)) ≈ false if ([] , ⟨⟩)

bool-theory : Theory bool-sig vars [ tt , tt , . . . ]
bool-theory = ⟨ commAnd , leastDef , . . . ⟩

pattern commAndAx = here
pattern leastDefAx = there here
```

4. Prove theorems using the axioms of the theory just defined. If a proof uses transitivity, one can use the equational reasoning idiom provided by the standard library of Agda:

```
p₁ : bool-theory ⊢ (⋀ ¬ p ∧ p ≈ false)
p₁ = begin
  ¬ p ∧ p
  ≈⟨ psubst commAndAx σ₁ ∼⟨⟩ ⟩
  p ∧ ¬ p
  ≈⟨ psubst leastDefAx idSubst ∼⟨⟩ ⟩
  false
  ∎
```

In the justification steps of this proof we use the substitution rule. The relevant actions of the substitution σ_1 are σ_1 p = ¬ p and σ_1 q = p.

4 Morphisms between signatures

In this section we explain our formalization of morphisms between signatures; this notion is interesting because it provides a conceptual understanding of syntactic translations. After pointing to some related works, we motivate the usefulness of this notion by showing a relatively simple example: how to interpret the Boolean theory of the previous section in the propositional calculus of Dijkstra and Scholten. [9]

The concept of morphism between signatures is related with the interpretability of similarity types in universal algebra (cf. [13]), and has an extensive literature: Fujiwara [12] introduced this notion as *mappings between algebraic systems*, Janssen [23], following the ADJ group, called it a *polynomial derivor* and Mossakowski et al. [26] referred to it as a *derived signature morphism*, a generalization of the more restricted *signature morphisms* in the theory of institutions [15].

Let us analyze how to translate the Boolean theory of the previous section to the propositional calculus of Dijkstra and Scholten [10], whose only non-constant operations are equivalence and disjunction.

```
data bool-ops' : List ⊤ × ⊤ → Set where
  f' t' : bool-ops' ([] ↦ tt)
  equiv' or' : bool-ops' ([ tt , tt ] ↦ tt)
bool-sig' : Signature
bool-sig' = record {sorts = ⊤ , ops = bool-ops'}
```

It is clear that one can translate recursively any term over bool-sig to a term in bool-sig' preserving its semantics. An alternative and more general way is to specify how to translate each operation in bool-sig using operations in bool-sig'. In this way, any bool-sig'-algebra can be seen as a bool-sig-algebra: a bool-sig-operation f

[9] Rocha and Meseguer [27] study more thoroughly Boolean theories and their morphisms.

$$\frac{}{[s_1, \ldots, s_n] \Vdash^{\Sigma} \sharp i : s_i}\ (\text{PRJ}) \qquad \frac{f : [s_1, ..., s_n] \Rightarrow_{\Sigma} s \ \ ar \Vdash^{\Sigma} t_1 : s_1 \ \cdots \ \ ar \Vdash^{\Sigma} t_n : s_n}{ar \Vdash^{\Sigma} f\,(t_1, ..., t_n) : s}\ (\text{OP})$$

Fig. 2. Type system for formal terms

is interpreted as the semantics of the translation of f. In particular, the translation of formulas is recovered as the initial homomorphism between |T| bool-sig and the transformation of |T| bool-sig'. In this section we formalize the concepts of *derived signature morphism* and *reduct algebra* as introduced, for example, by Sanella et al. [30].

4.1 Derived signature morphism

Although the disjunction from bool-sig can be directly mapped to its namesake in bool-sig', there is no unary operation in bool-sig' to translate the negation. In fact, we should be able to translate an operation as a combination of operations in bool-sig' and also refer to the arguments of the original operation.

We introduce the notion of *formal terms* which are formal composition of projections and operations. We introduce a type system, shown in Fig. 2, ensuring the well-formedness of these terms: the contexts are arities, i.e. lists of sorts, and identifiers are pointers (like de Bruijn indices). It can be formalized as an inductive family parameterized by arities and indexed by sorts.

```
data _⊩_ (ar' : Arity Σ) : (sorts Σ) → Set where
  #_  : (n : Fin (length ar')) → ar' ⊩ (ar' !! n)
  _|$|_ : ∀ {ar s} → ops Σ (ar ⇒ s) → HVec (ar' ⊩_) ar → ar' ⊩ s
```

A formal term specifies how to interpret an operation from the source signature in the target signature. The arity ar' specifies the sort of each argument of the original operation. For example, since the operation neg is unary, we can use one identifier when defining its translation. Notice that bool-sig and bool-sig' share the sorts; in general, one also considers a mapping between sorts.

A *derived signature morphism* consists of a mapping between sorts and a mapping from operations to formal terms:

```
record _↪_ (Σs Σt : Signature) : Set where
  field
    ↪s : sorts Σs → sorts Σt
    ↪o : ∀ {ar s} → ops Σs (ar , s) → (map ↪s ar) ⊩ (↪s s)
```

We show the action of the morphism on the operations neg and and

```
ops↪ : ∀ {ar s} → (f : bool-ops (ar ↦ s)) → map id ar ⊩ s
ops↪ neg = equiv' |$| ⟨p , f'⟩
ops↪ and = equiv' |$| ⟨equiv' |$| ⟨p , q⟩ , or' |$| ⟨p , q⟩⟩
```

where p = # zero and q = # (suc zero).

4.2 Transformation of Algebras

A signature morphism $m \colon \Sigma_s \hookrightarrow \Sigma_t$ induces a functor from Σ_t-algebras to Σ_s-algebras. Given a Σ_t-algebra $\mathcal{A}$, we denote with $\langle \mathcal{A} \rangle$ the corresponding Σ_s-algebra, which is known as the *reduct algebra with respect to the morphism m*. Let us sketch the construction of the functor on algebras: the interpretation of a Σ_s-sort s is given by $\langle \mathcal{A} \rangle_s = \mathcal{A}_{(m\,s)}$ and for interpreting an operation f in the reduct algebra $\langle \mathcal{A} \rangle$ we use the interpretation of the formal term mf, which is recursively defined by

```
⟦_⟧t : ∀ {ar s} → ar ⊩ s → ∥ A ⟦ ar ⟧s* ∥ → ∥ A ⟦ s ⟧s ∥
⟦ # n ⟧t as = as !!v n
⟦ f |$| ts ⟧t as = A ⟦ f ⟧o ⟨$⟩ ⟦ ts ⟧t* as
```

Identifiers denote projections and the application of the operation f to formal terms ts is interpreted as the interpretation of f applied to the denotation of each term in ts, the function ⟦_⟧$_t$* extends ⟦_⟧$_t$ to vectors.

We can formalize the reduct algebra in a direct way, however the interpretation of operations is a little more complicated, since we need to convince Agda that any vector vs : HVec (A ⟦_⟧$_s$ ∘ ↪$_s$) is has also the type HVec A (map ↪$_s$ is), which is accomplished by reindex-ing the vector (we omit the proof of cong):

```
module ReductAlg (m : Σs ↪ Σt) (A : Algebra Σt) where
  ⟨_⟩s : → (s : sorts Σs) → Setoid
  ⟨ s ⟩s = A ⟦ ↪s m s ⟧s

  ⟨_⟩o : ∀ {ar s} → ops Σs (ar ⇒ s) → (⟨_⟩s) * ar ≈⟶ ⟨ s ⟩s
  ⟨ f ⟩o = record { _⟨$⟩_ = ⟦ ↪o m f ⟧t ∘ reindex (↪s m); cong = ... }
  _⟨_⟩ : Algebra Σs
  _⟨_⟩ = record { _⟦_⟧s = ⟨_⟩s , _⟦_⟧o = ⟨_⟩o}
```

The action of the functor on homomorphisms is also straightforward.

A more interesting example of signature morphisms and reduct algebras is the definition of a compiler as presented in [33]. One defines a signature for the source language and another one for the target language; these languages are the term algebras over their respective signatures. A compiler is specified by a signature morphism from the source signature to the target signature: indeed the compiler is obtained as the unique homomorphism from the source algebra to the reduct algebra of the target algebra. Moreover, one can obtain a correct compiler by providing semantics of each language as algebras and a morphism between the source semantics and the reduct of the target semantics.[10]

[10] We explored this idea by defining a correct compiler for an arithmetic language targeting a stack-based language; it can be found at the repository in `Examples/CompilerArith.agda`.

4.3 Translation of theories

From a signature morphism $m : \Sigma_s \hookrightarrow \Sigma_t$ one gets the translation of ground Σ_s terms as the initial homomorphism from $|\mathsf{T}|\ \Sigma_s$ to $\langle\ |\mathsf{T}|\ \Sigma_t\ \rangle$. With an appropriate extension to variables, this translation applied to a theory E_s over Σ_s yields the theory $\widetilde{E_s}$ over Σ_t. Moreover if $\mathcal{A}_t \models \widetilde{E_s}$, one would think that the reduct $\langle \mathcal{A}_t \rangle$ is a model of the original theory, i.e. $\langle \mathcal{A}_t \rangle \models E_s$. Even better, if E_t is a stronger theory than the translated theory $\widetilde{E_s}$ and if $\mathcal{A}_t$ is a model for E_t, we would like that the reduct algebra models E_s. In Agda such a result would be realized as a function $\models\hookrightarrow$ with the following type (where $\hookrightarrow^*\ \mathsf{E}_s$ is the translation of E_s):

$$\models\hookrightarrow\ :\ \forall\ \mathsf{A}_t\ \mathsf{E}_t\ \mathsf{E}_s \rightarrow \mathsf{A}_t \models_m \mathsf{E}_t \rightarrow (\mathsf{E}_t \vdash\mathsf{T} \hookrightarrow^*\ \mathsf{E}_s) \rightarrow \langle\ \mathsf{A}_t\ \rangle \models_m \mathsf{E}_s$$

With the morphism $m : \Sigma_s \hookrightarrow \Sigma_t$, one can define the translation of open terms from $|\mathsf{T}|\ \Sigma_s\ (\!|\ \mathsf{X}_s\ |\!)$ to $|\mathsf{T}|\ \Sigma_t\ (\!|\ \mathsf{X}_t\ |\!)$ using initiality if we also have a renaming function $\hookrightarrow_v\ :\ \{\mathsf{s}\ :\ \mathsf{sorts}\ \Sigma_s\} \rightarrow \mathsf{X}_s\ \mathsf{s} \rightarrow \mathsf{X}_t\ (\mathsf{m} \hookrightarrow_s \mathsf{s})$. In general, however, we cannot prove the *satisfaction property*: if a Σ_t-algebra models the translation of an equation, then its reduct models the original equation. The technical issue is the impossibility of defining a Σ_t-environment from a Σ_s-environment. There is a well-known solution which consists on restricting the set of variable of the target signature by letting $X_t = \bigcup_{s \in \Sigma_s, t = m \hookrightarrow s} X_s$. Under this restriction, we can prove the satisfaction property and furthermore define the function $\models\hookrightarrow$. Such a restriction over the set of variables seems to us as an impediment, which can be alleviated if the original variables of E_t are included in the calculated set of variables.

5 Conclusions

As far as we know, heterogeneous universal algebra has not attracted a great interest in the academic community of type theory. In this paper, we have developed in Agda a library with the main concepts of heterogeneous universal algebra, up to the proof of the three isomorphisms theorems and the freeness of the term algebra over a set of variables. In order to define the term algebra we have introduced heterogeneous vectors, which later turned out to be very useful in other parts of the library, for example as the set of axioms of finite theories and as premises of deduction rules. We further introduced a formal system for conditional equational logic and proved its soundness and completeness with respect to Goguen and Meseguer semantics (we refer the reader to [34] for a deeper explanation of this result recasting it on a categorical setting). Finally, we defined a novel representation for (derived) signature morphisms and its associated contra-variant functor on algebras. We also showed that, under some restrictions, this functor also preserves models.

Related Work. Let us contrast our work with other formalizations covering some aspects of universal algebra. As far as we know, since Capretta's [7] first mechanization of universal algebra and its further extension to equational logic in his thesis, the closest new works are Kahl's [24] formalization of allegories and the development of the algebraic hierarchy lead by Spitters [31]. Capretta considered only finitary signatures and his work does not encompass signature morphisms. Spitters and his co-workers developed some very preliminary definitions of universal algebra, because their goal is to use the notion of variety to define the algebraic hierarchy up to the construction of the reals; in particular they use Coq's typeclasses to have a cleaner representation of algebraic structures.

Future Work. We think that this development opened the path to several further work, in particular: (i) a natural step is to formalize institutions;(ii) consider algebras of binding structures as proposed by Fiore [11], Capretta's and Felty's formalization [8] of higher-order algebras might be an interesting starting point;(iii) introduce multi-sorted rewriting;(iv) formalize more of the mathematical theory behind universal algebra, for example Birkhoff's (quasi)-variety characterization; and(v) explore the idea of using completeness and soundness for automating the proof of identities in algebraic structures.

Acknowledgement

We are grateful to the anonymous referees for their careful reading and suggestions.

References

[1] Barthe, G., V. Capretta and O. Pons, *Setoids in type theory*, J. Funct. Program. **13** (2003), pp. 261–293.

[2] Birkhoff, G., *On the structure of abstract algebras*, Mathematical Proceedings of the Cambridge Philosophical Society **31** (1935), p. 433–454.

[3] Birkhoff, G., "Lattice Theory," Colloquium Publications **25**, American Mathematical Society, 1940.

[4] Birkhoff, G. and J. D. Lipson, *Heterogeneous algebras*, J. of Combinatorial Theory **8** (1970), pp. 115–133.

[5] Bove, A., P. Dybjer and U. Norell, *A brief overview of agda - A functional language with dependent types*, in: *TPHOLs*, Lecture Notes in Computer Science **5674** (2009), pp. 73–78.

[6] Burstall, R. M., *Proving Properties of Programs by Structural Induction*, The Computer Journal **12** (1969), pp. 41–48.
URL http://dx.doi.org/10.1093/comjnl/12.1.41

[7] Capretta, V., *Universal algebra in type theory*, in: *International Conference on Theorem Proving in Higher Order Logics*, Springer, 1999, pp. 131–148.

[8] Capretta, V. and A. Felty, *Higher-order abstract syntax in type theory*, in: S. B. Cooper, H. Geuvers, A. Pillay and J. Väänänen, editors, *Logic Colloquium 2006*, Lecture Notes in Logic **32** (2009), pp. 65–90.

[9] Danielsson, N. A. and The Agda Team, *The agda standard library, version 0.12*, https://github.com/agda/agda-stdlib (2015).

[10] Dijkstra, E. W. and C. S. Scholten, "Predicate Calculus and Program Semantics," Springer New York, New York, NY, 1990.
URL http://dx.doi.org/10.1007/978-1-4612-3228-5

[11] Fiore, M. and O. Mahmoud, *Second-order algebraic theories*, in: *International Symposium on Mathematical Foundations of Computer Science*, Springer, 2010, pp. 368–380.

[12] Fujiwara, T., *On mappings between algebraic systems*, Osaka Math. J. **11** (1959), pp. 153–172.
URL http://projecteuclid.org/euclid.ojm/1200689635

[13] García, O. C. and W. Taylor, *The lattice of interpretability types of varieties*, Mem. Amer. Math. Soc. **50** (1984), pp. v+125.
URL http://dx.doi.org/10.1090/memo/0305

[14] Goguen, J. A., *Memories of ADJ*, Bulletin of the EATCS **39** (1989), pp. 96–102.

[15] Goguen, J. A. and R. M. Burstall, *Institutions: Abstract model theory for specification and programming*, J. ACM **39** (1992), pp. 95–146.
URL http://doi.acm.org/10.1145/147508.147524

[16] Goguen, J. A. and K. Lin, *Specifying, programming and verifying with equational logic.*, in: *We Will Show Them! Essays in Honour of Dov Gabbay, Volume Two* (2005), pp. 1–38.

[17] Goguen, J. A. and J. Meseguer, *Completeness of many-sorted equational logic*, SIGPLAN Notices **17** (1982), pp. 9–17.
URL http://doi.acm.org/10.1145/947886.947887

[18] Goguen, J. A. and J. Meseguer, *Remarks on remarks on many-sorted equational logic*, SIGPLAN Notices **22** (1987), pp. 41–48.

[19] Goguen, J. A., J. W. Thatcher, E. G. Wagner and J. B. Wright, *Abstract data types as initial algebras and the correctness of data representations*, in: *Conference on Computer Graphics, Pattern Recognition, & Data Structure, UCLA* (1975), pp. 89–93.

[20] Goguen, J. A., J. W. Thatcher, E. G. Wagner and J. B. Wright, *Initial algebra semantics and continuous algebras*, J. ACM **24** (1977), pp. 68–95.

[21] Gonthier, G., A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. L. Roux, A. Mahboubi, R. O'Connor, S. O. Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi and L. Théry, *A machine-checked proof of the odd order theorem*, in: *ITP*, Lecture Notes in Computer Science **7998** (2013), pp. 163–179.

[22] Huet, G. and D. C. Oppen, *Equations and rewrite rules: a survey*, Technical Report STAN//CS-TR-80-785, Stanford University, Department of Computer Science (1980).

[23] Janssen, T. M., *Algebraic translations, correctness and algebraic compiler construction*, Theoretical Computer Science **199** (1998), pp. 25–56.

[24] Kahl, W., *Dependently-typed formalisation of relation-algebraic abstractions*, in: *RAMICS*, Lecture Notes in Computer Science **6663** (2011), pp. 230–247.

[25] Meinke, K. and J. V. Tucker, *Universal algebra*, in: S. Abramsky and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science (Vol. 1)*, Oxford University Press, Inc., New York, NY, USA, 1992 pp. 189–368.

[26] Mossakowski, T., U. Krumnack and T. Maibaum, *What is a derived signature morphism?*, in: *Revised Selected Papers of the 22Nd International Workshop on Recent Trends in Algebraic Development Techniques - Volume 9463*, WADT 2014 (2015), pp. 90–109.

[27] Rocha, C. and J. Meseguer, *Five isomorphic boolean theories and four equational decision procedures*, Technical report, University of Illinois at Urbana-Champaign (2007).

[28] Rocha, C. and J. Meseguer, *Theorem proving modulo based on boolean equational procedures*, in: *RelMiCS*, Lecture Notes in Computer Science **4988** (2008), pp. 337–351.

[29] Salvesen, A. and J. M. Smith, *The strength of the subset type in Martin-Löf's type theory*, in: *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988* (1988), pp. 384–391.
URL https://doi.org/10.1109/LICS.1988.5135

[30] Sannella, D. and A. Tarlecki, "Foundations of algebraic specification and formal software development," Springer Science & Business Media, 2012.

[31] Spitters, B. and E. van der Weegen, *Type classes for mathematics in type theory*, Mathematical Structures in Computer Science **21** (2011), pp. 795–825.

[32] Tarlecki, A., *Some nuances of many-sorted universal algebra: A review*, Bulletin of the EATCS **104** (2011), pp. 89–111.
URL http://albcom.lsi.upc.edu/ojs/index.php/beatcs/article/view/79

[33] Thatcher, J. W., E. G. Wagner and J. B. Wright, *More on advice on structuring compilers and proving them correct*, Theoretical Computer Science **15** (1981), pp. 223–249.

[34] Vidal, J. C. and J. S. Tur, *On the completeness theorem of many-sorted equational logic and the equivalence between Hall Algebras and Bénabou theories*, Reports on Mathematical Logic **40** (2006), pp. 127–158.
URL http://www.iphils.uj.edu.pl/rml/rml-40/06-climent.pdf