Universal Algebra in Type Theory

Venanzio Capretta

Computer Science Institute
University of Nijmegen
venanzio@cs.kun.nl

Abstract. We present a development of Universal Algebra inside Type Theory, formalized using the proof assistant Coq. We define the notion of a signature and of an algebra over a signature. We use setoids, i.e. types endowed with an arbitrary equivalence relation, as carriers for algebras. In this way it is possible to define the quotient of an algebra by a congruence. Standard constructions over algebras are defined and their basic properties are proved formally. To overcome the problem of defining term algebras in a uniform way, we use types of trees that generalize wellorderings. Our implementation gives tools to define new algebraic structures, to manipulate them and to prove their properties.

1 Introduction

The development of mathematical theories inside Type Theory presents some technical problems that make it difficult to translate an informal mathematical proof into a formalized one. In trying to carry out such a translation, one soon realizes that notions that were considered non-problematic and obvious at the informal level need a delicate formal analysis. Additional work is often needed just to define the mathematical structures under study and the basic tools to manipulate them. Besides the difficulty of rendering exactly what is expressed only in intuitive terms, there is the non-trivial task of translating into Type Theory what was originally intended to be expressed inside some form of set theory (for example in ZF). This paper presents a development of such tools for generic algebraic reasoning, which has been completely formalized in the Coq proof development system (see [3]). We want to enable the users of such tools to easily define their own algebraic structures, manipulate objects and reason about them in a way that is not too far from ordinary mathematical practice.

Our work stemmed from an original project of formal verification of Computer Algebra algorithms in Type Theory. We realized then that the definition of common mathematical structures, like those of ring and field, together with tools to manipulate them, was essential to the success of the enterprise. We decided to develop Universal Algebra as a general tool to define algebraic structures.

Previous work on Algebra in Type Theory was done by Paul Jackson using the proof system Nuprl (see [14]), by Peter Aczel on Galois Theory (see [1]) and by Huet and Saïbi on Category Theory (see [13]). A large class of algebraic structures has been developed in Coq by Loïc Pottier.

Another aim is the use of a two level approach to the derivation of propositions about algebraic objects (see [4]). In this approach, statements about objects are lifted to a syntactic level where they can be manipulated by operators. An example is the simplification of expressions and automatic equational reasoning. This method was already used by Douglas Howe to construct a partial syntactic model of the Type Theory of Nuprl inside Nuprl itself, which can be used to program tactics inside the system (see [12]). An application of this reflection mechanism to algebra was developed by Samuel Boutin in Coq for the simplification of ring expressions (see [5]). In the present work the need to parameterize the construction of the syntactic level on the type of signatures posed an additional problem. A very general type construction similar to Martin-Löf's Wellorderings was employed for the purpose.

Finally, the study of the computational content of algebras is particularly interesting. We investigate to what extent algebraic objects can be automatically manipulated inside a proof checker. This can be done through the use of certified versions of algorithms borrowed from Computer Algebra, as was done by Théry in [23] and by Coquand and Persson in [8] for Buchberger's algorithm.

The files of the implementation are available via the Internet at the site *http://www.cs.kun.nl/~venanzio/universal_algebra.html.*

Type Theory and Coq. The work presented here has been completely formalized inside Coq, but it could have equally easily been formalized in other proof systems based on Type Theory, like Lego or Alf. Although Coq is based on the Extended Calculus of Constructions (see [15]) everything could be formalized in a weaker system. Any Pure Type System that is at least as expressive as $\lambda P\overline{\omega}$ (see [2]) endowed with inductive types (see [22]), or Martin-Löf's Type Theory with at least two universes (see [16], [17] or [19]) is enough.

We assume that we have two universes of types $*^s$ for sets and $*^p$ for propositions (`Set` and `Prop` in the syntax of Coq), and that they both belong to the higher universe $\Box$ (`Type` in Coq). The product type $\Pi x : A.B$ is written using Coq notation $(x : A)B$. If $B : *^p$ we also write $(\forall x : A)B$. In Coq it is possible to define record types in which every field can depend on the values of the preceding fields. We will use the following notation for records.

$$Record\ Name\ : Type\ :=\ constructor \begin{cases} field_1 : A_1 \\ \vdots \\ field_n : A_n \end{cases}$$

An element of this record type is in the form $(constructor\ a_1 \ldots a_n)$ where $a_1 : A_1, a_2 : A_2[field_1 := a_1], \ldots, a_n : A_n[field_1 := a_1, \ldots, field_{n-1} := a_{n-1}]$. We have the projections

$$\begin{array}{l} field_1 : Name \to A_1 \\ field_2 : (x : Name)(A_2[field_1 := (field_1\ x)]) \\ \vdots \\ field_n : (x : Name)(A_n[field_1 := (field_1\ x)] \ldots [field_{n-1} := (field_{n-1}\ x)]) \end{array}$$

In Coq a record type is a shorthand notation for an inductive type with only one constructor. In a system without this facility they could be represented by nested Σ-types.

Algebraic structures in Type Theory. Let us start by considering a simple algebraic structure and its implementation in Type Theory. The standard mathematical definition of a group is the following.

Definition 1. *A* group *is a quadruple* $\langle G, *, e, \_^{-1}\rangle$*, where* G *is a set,* $*$ *a binary operation on* G*,* e *an element of* G *and* $\_^{-1}$ *a unary operation on* G *such that* $(x * y) * z = x * (y * z)$*,* $x * e = x$ *and* $x * (x^{-1}) = e$ *for all* $x, y, z \in G$*.*

An immediate translation in Type Theory would employ a record type.

$$\begin{array}{l} \mathit{Record\ Group}\ : \Box\ := \\ \quad \mathit{group} \left\{ \begin{array}{ll} \mathit{elements} & : \mathit{Setoid} \\ \mathit{operation} & : \mathit{elements} \rightarrow \mathit{elements} \rightarrow \mathit{elements} \\ \mathit{unit} & : \mathit{elements} \\ \mathit{inverse} & : \mathit{elements} \rightarrow \mathit{elements} \end{array} \right. \end{array}$$

where a setoid is a set endowed with an equivalence relation (see next section).

But this is not yet enough since we didn't specify that the group axioms must be satisfied. This is usually done by enlarging the record to contain proofs of the axioms.

$$\begin{array}{l} \mathit{Record\ Group}\ : \Box\ := \\ \quad \mathit{group} \left\{ \begin{array}{ll} \mathit{elements} & : \mathit{Setoid} \\ \mathit{operation} & : \mathit{elements} \rightarrow \mathit{elements} \rightarrow \mathit{elements} \\ \mathit{unit} & : \mathit{elements} \\ \mathit{inverse} & : \mathit{elements} \rightarrow \mathit{elements} \\ \mathit{associativity} & : (\forall x, y, z : \mathit{elements}) \\ & \quad (\mathit{operation}\ (\mathit{operation}\ x\ y)\ z) \\ & \quad = (\mathit{operation}\ x\ (\mathit{operation}\ y\ z)) \\ \mathit{unitax} & : (\forall x : \mathit{elements})(\mathit{operation}\ x\ \mathit{unit}) = x \\ \mathit{inverseax} & : (\forall x : \mathit{elements})(\mathit{operation}\ x\ (\mathit{inverse}\ x)) = \mathit{unit} \end{array} \right. \end{array}$$

So to declare a specific group, for example the group of integers with the sum operation, we must specify all the fields:

$$\mathit{Integer} : \mathit{Group} := (\mathit{group}\ Z\ \mathit{plus}\ 0\ -\ p1\ p2\ p3).$$

where $p1$, $p2$, $p3$ are proofs of the axioms.

Why it is useful to develop Universal Algebra. Once an algebraic structure has been specified in this way, we proceed to give standard definitions like those of subgroup, product of groups, quotient of a group by a congruence relation, homomorphism of groups and we prove standard results. In this way many algebraic structures can be specified, and theorems can be proved about them (see the work by Loïc Pottier).

Since most of the definitions and basic properties are the same for every algebraic structure, having an abstract general formulation of them would save us from duplicating the same work many times. This is the main reason why it is interesting to develop Universal Algebra. To this aim we should internalize the generalization of the previous construction to have a general notion of algebraic structure inside Type Theory.

2 Setoids

Why we need setoids, informal definition of setoid. The first step before the implementation of Universal Algebra in Type Theory is to have a flexible translation of the intuitive notion of set. Interpreting sets as types would rise some problems: the structure of types is rather rigid and does not allow the formation of subtypes or quotient types. Since we need to define subalgebras and quotient algebras we are led to consider a more suitable solution. In some version of (extensional) type theory notions of subtype and quotient type are implemented (for example in the Nuprl system, see [6]), but the version of (intensional) type theory implemented in Coq does not. Nevertheless a model of extensional type theory inside intensional type theory has been constructed by Martin Hofmann (see [10]). We use a variant of this model, which has already been implemented by Huet and Saïbi in [13] and used by Pottier.

The elements of a type are build up using some constructors, and elements of a type are said to be equal when they are convertible. Thus a type cannot be defined by a predicate over an other type (subtyping) or by redefining the equality (quotienting). We allow ourselves to be more liberal with equality by defining a setoid to be a pair formed by a set and an equivalence relation over it. Thus we can quotient a setoid by just changing the equivalence relation. Subsetoids are obtained by quotienting Σ-types, i.e. if $\mathcal{S} = \langle A, =_{\mathcal{S}} \rangle$ is a setoid and P is a predicate over A (that is closed under $=_{\mathcal{S}}$), we can define the subsetoid determined by P to be $\mathcal{S}^P = \langle (\Sigma x : A.(P\ x)), =_{\mathcal{S}^P} \rangle$ where $\langle a_1, p_1 \rangle =_{\mathcal{S}^P} \langle a_2, p_2 \rangle$ iff $a_1 =_{\mathcal{S}} a_2$.

Since we explicitly work with equivalence relations all the definitions on setoids (predicates over setoids, relations between setoids, setoids functions) must be required to be invariant under the given equality.
Formal definition of setoid.

Definition 2.

$$Record\ Setoid\ : \Box\ :=\ setoid \begin{cases} s\_el & : *^s \\ s\_eq & : s\_el \to s\_el \to *^p \\ s\_proof & : (Equiv\ s\_eq) \end{cases}$$

where $(Equiv\ s\_eq)$ *is the proposition stating that* s_eq *is an equivalence relation over the set* s_el.

We often identify a setoid $\mathcal{S}$ with its carrier set $(s\_el\ \mathcal{S})$. In Coq this identification is realized through the use of *implicit coercions* (see [21]). Similar implicit coercions are also used to identify an algebraic structure with its carrier. If $a, b : \mathcal{S}$ (i.e. as we said $x, y : (s\_el\ \mathcal{S})$), we use the simple notation $x = y$ in place of

$(s\_eq\ x\ y)$; in Coq an infix operator `[=]` is defined so we can write `x [=] y`. As a general methodology if `op` is a set operator, we use the notation `[op]` for the corresponding setoid operator. Whenever we want to stress the setoid in which the equality holds (two setoids may have the same elements but different equalities) we write $x =_{\mathcal{S}} y$.

Properties and constructions on setoids. As we have mentioned above, we have to be careful when dealing with constructions on setoids. For example, predicates, relations and functions should be invariant under the given equality.

Definition 3. *A predicate P over the carrier of a setoid $\mathcal{S}$, i.e. $P : (s\_el\ \mathcal{S}) \to *^p$ is said to be* well defined *(with respect to $=_{\mathcal{S}}$) if* $(\forall x, y : \mathcal{S}) x =_{\mathcal{S}} y \to (P\ x) \to (P\ y)$. *The type of* setoid predicates *over $\mathcal{S}$ is the record type*

$$\begin{array}{l} Record\ Setoid\_predicate\ : \Box\ := \\ \quad setoid\_predicate \begin{cases} sp\_pred & : \mathcal{S} \to *^p \\ sp\_proof & : (Predicate\_well\_defined\ sp\_pred) \end{cases} \end{array}$$

where $(Predicate\_well\_defined\ sp\_pred)$ is the above property.

Definition 4. *A relation R on the carrier of a setoid $\mathcal{S}$, i.e. $P : (s\_el\ \mathcal{S}) \to (s\_el\ \mathcal{S}) \to *^p$ is said to be* well defined *(with respect to $=_{\mathcal{S}}$) if*

$$(\forall x_1, x_2, y_1, y_2 : \mathcal{S}) x_1 =_{\mathcal{S}} x_2 \to y_1 =_{\mathcal{S}} y_2 \to (R\ x_1\ y_1) \to (R\ x_2\ y_2).$$

The type of setoid relations *on $\mathcal{S}$ is the record type*

$$\begin{array}{l} Record\ Setoid\_relation\ : \Box\ := \\ \quad setoid\_relation \begin{cases} sr\_rel & : \mathcal{S} \to \mathcal{S} \to *^p \\ sr\_proof & : (Relation\_well\_defined\ sr\_rel) \end{cases} \end{array}.$$

By declaring two implicit coercions we can use setoid predicates and relations as if they were regular predicates and relations, i.e. if $P : (Setoid\_predicate\ \mathcal{S})$ and $x : \mathcal{S}$ then $(P\ x)$ is a shorthand notation for $((sp\_pred\ P)\ x) : *^p$ and if $R : (Setoid\_relation\ \mathcal{S})$ and $x, y : \mathcal{S}$ then $(R\ x\ y)$ is a shorthand notation for $((sr\_rel\ R)\ x\ y) : *^p$.

As we have mentioned in the informal discussion subsetoids can be defined by a setoid predicate by giving a suitable equivalence relation over a Σ-type.

Definition 5. *Let $\mathcal{S}$ be a setoid and P a setoid predicate over it. Then* the subsetoid of $\mathcal{S}$ separated by P *is the setoid $\mathcal{S}|P$ that has carrier $\Sigma x : \mathcal{S}.(P\ x)$ and equality relation $\langle a_1, p_1 \rangle =_{\mathcal{S}|P} \langle a_1, p_1 \rangle \iff a_1 =_{\mathcal{S}} a_2$.*

Even easier is the definition of a quotient of a setoid by an equivalence (setoid) relation. It is enough to substitute such relation in place of the original equality.

Definition 6. *Let $\mathcal{S}$ be a setoid and $Eq : (Setoid\_relation\ \mathcal{S})$ such that $(sr\_rel\ Eq)$ is an equivalence relation on $(s\_el\ \mathcal{S})$. Then the* quotient setoid *$\mathcal{S}/Eq$ is the setoid with carrier set $(s\_el\ \mathcal{S})$ and equality relation Eq.*

Notice that the notion of quotient setoid is different from the notion of quotient set in set theory: the elements of $\mathcal{S}/Eq$ are not equivalence classes, as in set theory, but they are exactly the same as the elements of $\mathcal{S}$.

Definition 7. *Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be two setoids. Their* product *is the setoid $\mathcal{S}_1[\times]\mathcal{S}_2$ with carrier set $(s\_el\ \mathcal{S}_1) \times (s\_el\ \mathcal{S}_2)$ and equality relation*

$$\langle x_1, x_2\rangle =_{\mathcal{S}_1[\times]\mathcal{S}_2} \langle y_1, y_2\rangle \iff x_1 =_{\mathcal{S}_1} y_1 \wedge x_2 =_{\mathcal{S}_1} y_2.$$

Definition 8. *Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be two setoids. The setoid of functions from $\mathcal{S}_1$ to $\mathcal{S}_2$ is the setoid $\mathcal{S}_1[\rightarrow]\mathcal{S}_2$ with carrier set the type of those functions between the two carriers that are well-defined with respect to the setoid equalities*

$$Record\ \mathcal{S}_1[\rightarrow]\mathcal{S}_2 \quad := \quad setoid\_function \begin{cases} s\_function \quad : \mathcal{S}_1 \rightarrow \mathcal{S}_2 \\ s\_fun\_proof : \\ \quad (fun\_well\_defined\ s\_function) \end{cases}$$

where $(fun\_well\_defined\ s\_function)$ is the proposition $(\forall x_1, x_2 : \mathcal{S}_1)x_1 =_{\mathcal{S}_1} x_2 \rightarrow (s\_function\ x_1) =_{\mathcal{S}_2} (s\_function x_2)$, and $f =_{\mathcal{S}_1[\rightarrow]\mathcal{S}_2} g$ is the extensional equality relation $(\forall x : \mathcal{S}_1)(f\ x) =_{\mathcal{S}_2} (g\ x)$.

In a similar way we can define other constructions on setoids and define operators on them (see the source files for a complete list).

3 Signatures and Algebras

Using the development of setoids from the previous section as our notion of sets we can now translate Universal Algebra into Type Theory. We use as a guide the chapter on Universal Algebra by K. Meinke and J. V. Tucker from the *Handbook of Logic in Computer Science* ([18]). We differ from that work only in that we consider just finite signatures (so that they can be implemented by lists) and we do not require that carrier sets are non-empty. This second divergence is justified by the difference between first order predicate logic (which is the logic usually employed to reason about algebraic structures), that always assumes the universe of discourse to be non-empty, and Type Theory, in which this assumption is not present and, therefore, we can reason about empty structures (about this see also [2], section 5.4).

Definition of signature. We begin by defining the notion of a (many-sorted) signature. A signature is an abstract specification of the carrier sets (called *sorts*) and operations of an algebra, and it is given by the number of sorts n and a list of operation symbols $[f_1, \ldots, f_m]$ where each of the functions f_i must be specified by giving its type, i.e. by saying how many arguments the function has, to which one of the sorts each argument belongs and to which sort the result of the application of the operation belongs. Each sort is identified by an element of the finite set $\mathbb{N}_n = \{0, \ldots, n-1\}$ (in our Coq implementation $\mathbb{N}_n$ is represented by `(Finite n)` and its elements are represented by `n}-(0)`, `n}-(1)`, $\ldots$, `n}-(n-1)`).

As an example suppose we want to define a structure $\langle nat, bool; O, S, true, false, eq\rangle$ to model the natural numbers and booleans together with a test function for equality with boolean values. So we want that

$$\begin{array}{llll} nat, bool & : Setoid & & \\ O & : nat & true, false & : bool \\ S & : nat \to nat & eq & : nat \times nat \to bool \end{array}$$

So in this case $n = 2$, the index of the sort *nat* is 0, the index of the sort *bool* is 1, and the types of constants and functions are

O	$\Rightarrow$	$\langle [], 0\rangle$	(no arguments and result in *nat*)
S	$\Rightarrow$	$\langle [0], 0\rangle$	(one argument from *nat*, result in *nat*)
$true$	$\Rightarrow$	$\langle [], 1\rangle$	(no arguments, result in *bool*)
$false$	$\Rightarrow$	$\langle [], 1\rangle$	(no arguments, result in *bool*)
eq	$\Rightarrow$	$\langle [0, 0], 1\rangle$	(two arguments from *nat*, result in *bool*)

Definition 9. *Let $n : \mathbb{N}$ be a fixed natural number. Let $Sort \equiv \mathbb{N}_n$. A* function type *is a pair $\langle args, res\rangle$, where args is a list of elements of Sort (indicating the type of the arguments of the function) and res is an element of Sort (indicating the type of the result). So in Type Theory we define the type of function types as $(Function\_type\ n) := (list\ Sort) \times Sort$.*

Definition 10. *A* signature *is a pair $\langle n, fs\rangle$ where $n : \mathbb{N}$ and $fs \equiv [f_1, \ldots, f_m]$ is a list of function types. We represent it in Type Theory by a record type:*

$$\begin{array}{l} Record\ Signature\ : *^s\ := \\ \quad signature \begin{cases} sorts\_num & : \mathbb{N} \\ function\_types & : (list\ (Function\_type\ sorts\_num)) \end{cases} \end{array}$$

The signature of natural numbers and booleans is then defined as $\sigma = (signature\ 2\ [\langle [], 0\rangle, \langle [0], 0\rangle, \langle [], 1\rangle, \langle [], 1\rangle, \langle [0, 0], 1\rangle])$.

Definition of algebra. Let $\sigma : Signature$, we want to define the notion of a σ-algebra. To define such a structure we need to interpret the sorts as setoids, and the function types as setoid functions. Suppose $\sigma = \langle n, [f_1, \ldots, f_s]\rangle$. The interpretation of the sorts is a family of n setoids: $Sorts\_interpretation := \mathbb{N}_n \to Setoid$. So let us assume that $sorts : Sorts\_interpretation$, and define the interpretation of $f_1, \ldots, f_n$. There are several ways of defining the type of a function, depending on how the arguments are given. Suppose $f = \langle [a_1, \ldots, a_k], r\rangle$ is a function type. If $x_j : (sorts\ a_j)$ for $j = 1, \ldots, k$, then we would like the interpretation of f, $\|f\|$, to be applicable directly to its arguments, $(\|f\|\ x_1\ \ldots x_k) : (sorts\ r)$. This means that $\|f\|$ should have the curried type $(sorts\ a_1)[\to] \ldots [\to](sorts\ a_k)[\to](sorts\ r)$ This type may be defined by using a general construction to define types of curried functions with arity and types of the arguments as parameters. This is done by the function

$$Curry\_type\_setoid : (n : nat)(\mathbb{N}_n \to Setoid) \to Setoid \to Setoid$$

such that if n is a natural number, $A : \mathbb{N}_n \to \mathit{Setoid}$ is a family of setoids defining the type of the arguments, and $B : \mathit{Setoid}$ is the type of the result, then

$$(\mathit{Curry\_type\_setoid}\ n\ A\ B) = (A\ 0)[\to] \ldots [\to](A\ n-1)[\to]B$$

So in the previous example the type of $\|f\|$ may be defined as

$$(\mathit{Curry\_type\_setoid}\ k\ [i : \mathbb{N}_k](\mathit{sorts}\ a_i)\ (\mathit{sorts}\ r))$$

But this representation is difficult to use when reasoning abstractly about functions, e.g. if we want to prove general properties of the functions which do not depend on the arity. In this situation it is better to see the function having just one argument containing all the x_j's. We can do that by giving the arguments as k-tuples or as functions indexed on a finite type. We choose this second option. So we represent the arguments as an object of type $(j : \mathbb{N}_k)(\mathit{sorts}\ a_j)$. Then the interpretation of the function f could have the type $((j : \mathbb{N}_k)(\mathit{sorts}\ a_j)) \to (\mathit{sorts}\ r)$. This is still not completely correct. Since the sorts are setoids, the interpretation of the functions must preserve the setoid equality. With the aim of formulating this condition, we first make the type of arguments $(j : \mathbb{N}_k)(\mathit{sorts}\ a_j)$ into a setoid by stating that two elements $\mathit{args}_1, \mathit{args}_2$ are equal if they are extensionally equal.

Definition 11. *Let $k : \mathbb{N}$, $A : \mathbb{N}_k \to \mathit{Setoid}$. Then $(\mathit{FF\_setoid}\ k\ A)$ is the setoid that has carrier $(j : \mathbb{N}_k)(A\ j)$ and equality relation*

$$(\mathit{args}_1 =_{(\mathit{FF\_setoid}\ k\ A)} \mathit{args}_2) \iff (\forall j : \mathbb{N}_k)((\mathit{args}_1\ j) =_{(A\ j)} (\mathit{args}_2\ j))$$

We can now interpret a function type and a list of function types.

Definition 12. *Let $f = \langle [a_1, \ldots, a_k], r \rangle$ be a function type. Then*

$$\begin{array}{l}(\mathit{Function\_type\_interpretation}\ n\ \mathit{sorts}\ f) := \\ (\mathit{FF\_setoid}\ k\ [i : \mathbb{N}_k](\mathit{sorts}\ a_k))[\to](\mathit{sorts}\ r)\end{array}$$

A list of function types is interpreted by the operator

$$\begin{array}{l}(\mathit{Function\_list\_interpretation}\ n\ \mathit{sorts}) : \\ (\mathit{list}\ (\mathit{Function\_type}\ n)) \to \mathit{Setoid}\end{array}$$

where the carrier of $(\mathit{Function\_list\_interpretation}\ n\ \mathit{sorts}\ [f_1, \ldots, f_s])$ is

$$[i : \mathbb{N}_s](\mathit{Function\_type\_interpretation}\ n\ \mathit{sorts}\ f_i)$$

(we do not need to take into consideration how the equality relation is defined).

This is the way in which functions are represented in the algebra. Whenever we want to have them in the curried form we can apply a conversion operator

$$\mathit{fun\_arg\_to\_curry} : ((\mathit{FF\_setoid}\ k\ A)[\to]B) \to (\mathit{Curry\_type\_setoid}\ k\ A\ B).$$

The inverse conversion is performed by the operator $\mathit{curry\_to\_fun\_arg}$.

Eventually, the type of σ-algebras can be defined as

Definition 13. *The type of* algebras over the signature σ *is the record type*

$$\begin{array}{l} Record\ (Algebra\ \sigma)\ :\Box\ := \\ \quad algebra \left\{ \begin{array}{ll} sorts & : (Sorts\_interpretation\ (sorts\_num\ \sigma)) \\ functions & : (Function\_list\_interpretation \\ & \quad (sorts\_num\ \sigma)\ sorts\ (function\_types\ \sigma)) \end{array} \right. \end{array}$$

The type of arguments corresponding to the i-th function of the signature σ in an algebra $\mathcal{A}$ are also indicated by $(Fun\_arg\_arguments\ \mathcal{A}\ i)$.

If $\sigma \equiv \langle n, [f_0, \ldots, f_{m-1}]\rangle$ and $\mathcal{A} : (Algebra\ \sigma)$, we indicate by $f_{i\mathcal{A}}$ the interpretation of the ith function symbol $f_{i\mathcal{A}} \equiv (functions\ \sigma\ \mathcal{A}\ i)$ for every $i : \mathbb{N}_m$. As an example let us define a σ-algebra for the signature considered before, interpreting the two sorts as the setoids of natural numbers and booleans (in these cases the equivalence relation is trivially Leibniz equality). Suppose we have already defined

$$\begin{array}{lll} Nat, Bool & : Setoid & \\ 0 & : Nat & T, F : Bool \\ S & : Nat[\rightarrow]Nat & Eq\ \ : Nat[\rightarrow]Nat[\rightarrow]Bool \end{array}$$

Then we can give the interpretation of the sorts

$$\begin{array}{l} Srt : (Sorts\_interpretation\ 2) \\ (Srt\ 0) = Nat \qquad (Srt\ 1) = Bool \end{array}$$

and of the functions

$$\begin{array}{ll} Fun : (Function\_list\_interpretation\ Srt\ (function\_types\ sigma)) & \\ (Fun\ 0) = (curry\_to\_fun\_arg\ 0) & (Fun\ 3) = (curry\_to\_fun\_arg\ F) \\ (Fun\ 1) = (curry\_to\_fun\_arg\ S) & (Fun\ 4) = (curry\_to\_fun\_arg\ Eq) \\ (Fun\ 2) = (curry\_to\_fun\_arg\ T) & \end{array}$$

Then we can define the σ-algebra

$$nat\_bool\_alg := (algebra\ \sigma\ Srt\ Fun) : (Algebra\ \sigma)$$

4 Term Algebras

Informal definition of term algebras. A class of algebras of special interest is that of Term Algebras. The sorts of such an algebra are the terms freely generated by the function symbols of the signature. For example, in the signature defined above we would have that the expressions O, $S(O)$, $S(S(O))$ are terms of the first sort, while *true*, *false*, $eq(O, S(O))$ are terms of the second. In general given a signature $\sigma = \langle n, [f_1, \ldots, f_m]\rangle$, the algebra of terms have carriers T_i, for $i : \mathbb{N}_n$, whose elements have the form $f_j(t_1, \ldots, t_k)$ where $j : \mathbb{N}_m$, the type of f_j is $\langle [a_1, \ldots, a_k], r\rangle$, $t_1, \ldots, t_k$ belong to the term sorts $T_{a_1}, \ldots, T_{a_k}$ respectively, and the resulting term is in the sort T_r.

Similarly we can define an algebra of open terms or expressions, i.e. terms in which variables can appear. We start by a family of sets of variables X_i for $i : \mathbb{N}_n$, and we construct terms by application of the function symbols as before.
Problem: the uniform definition. In Type Theory this can be easily modeled by inductively defined types whose constructors correspond to the functions of the signature. For example, the sorts of terms of the previous signature are the (mutually) inductive types

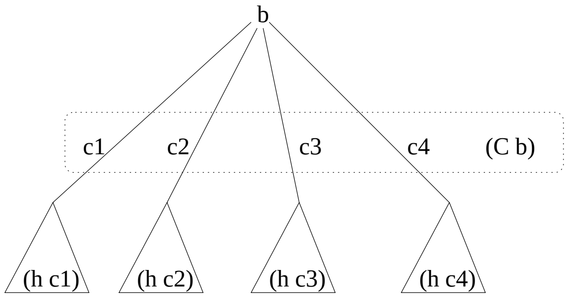
$$\begin{array}{rll} nat\_term := & o\_symb & : nat\_term \\ & \mid s\_symb & : nat\_term \to nat\_term \\ bool\_term := & t\_symb & : bool\_term \\ & \mid f\_symb & : bool\_term \\ & \mid eq\_symb & : nat\_term \to nat\_term \to bool\_term \end{array}$$

If the signature is single-sorted, a simple inductive definition gives the type of terms; if it is many-sorted then we have to use mutually inductive definitions. In this way we can define the types of sorts for any specific signature, but it is not possible to define it parametrically. We would like to define term algebras as a second order function

$$Term\_algebra : (\sigma : Signature)(Algebra\ \sigma)$$

that associates the corresponding term algebra to each signature. In order to do this we would need mutually inductive definitions in which the number of sorts and constructors and the type of the constructors are parametric. Such a general form of inductive definition is not available in current implementations of Type Theory (like Coq), so we have to look for a different solution.
Discussion on possible solutions. The problem is more general and regards the definition of families of inductive types in which every element of the family is a correct inductive type, but the family itself cannot be defined. In the general case we have a family of set operators indexed on a set A, $\Phi : A \to (*^s \to *^s)$ and we want to define a family of inductive types each of which is the minimal fixed point of the corresponding operator, i.e. we want a family $I : A \to *^s$ such that for every $a : A$, $(I\ a)$ is the minimal fixed point of $(\Phi\ a)$. In Type Theory it is possible to define the minimal fixed point of a set operator $\Psi : *^s \to *^s$ if and only if the set operator is strictly positive, i.e. in the expression $(\Psi\ X)$, where $X : *^s$, X occurs only to the right of arrows. But it may happen that even if for every concrete element (closed term) a of the set A, the operator $(\Phi\ a)$ is strictly positive or reduces to a strictly positive operator, this does not hold for open terms, i.e. if $x : A$ is a variable $(\Phi\ x)$ does not satisfy the strict positivity condition. There are several possibilities to overcome this difficulty. A thorough analysis of this subject will be the argument of a future paper. Here we adopt a solution that represents every inductive type by a type of trees.
Solution using Wellorderings. W types are a type theoretic implementation or the notion of well orderings as well-founded trees. They were introduced by Per Martin-Löf in [16] (see also [17] and [19], chapter 15). Suppose that we want to define a type of trees such that the nodes of the trees are labeled by

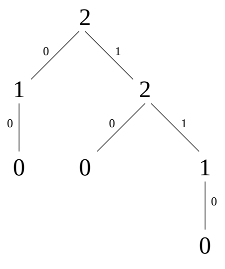
elements of the type B, and for each node labeled by an element $b : B$, the branches stemming from the node are labeled by the elements of a set $(C\ b)$, i.e. the b-node has as many branches as the elements of $(C\ b)$. Then the W type constructor has two parameters: a type $B : *^s$ and a family of types $C : B \to *^s$. To define a new element of the type $(W\ B\ C)$ we have to specify the label of the root by an element $b : B$ and for each branch, i.e. for every element $c : (C\ b)$, the corresponding subtree; this is done by giving a function $h : (C\ b) \to (W\ B\ C)$.



Formally we can define $(W\ B\ C)$ in the Calculus of Inductive Constructions (see [7] and [9]) as the inductive type $(W\ B\ C)$ with one constructor sup : $(b : B)((C\ b) \to (W\ B\ C)) \to (W\ B\ C)$. As for any inductive definition, we automatically get principles of recursion and induction associated with the definition. If $(C\ b)$ is infinite for some $b : B$ we get transfinite induction.

We can use this construction to define term algebras for single-sorted signatures, representing a term by its syntax tree. We choose B to be the set of function symbols of the signature (or just $\mathbb{N}_m$ where m is the number of the functions), and $(C\ f) = \mathbb{N}_{k_f}$ where k_f is the arity (number of arguments) of the function symbol f.

For example, let us take the signature $\langle 1, [\langle [], 0\rangle, \langle [0], 0\rangle, \langle [0,0], 0\rangle]\rangle$ describing a structure with one sort, one constant, one unary operation and one binary operation. Let us indicate the three functions by f_0 (the constant), f_1 (the unary operation) and f_2 (the binary operation). The type of terms is represented by the type $(W\ \mathbb{N}_3\ C)$ where $C = [i : \mathbb{N}_3](cases\ i\ of\ 0 \Rightarrow \mathbb{N}_0 | 1 \Rightarrow \mathbb{N}_1 | 2 \Rightarrow \mathbb{N}_2)$. Then the term $f_2(f_1(f_0), f_2(f_0, (f_1(f_0))))$ is represented by the tree



or formally by the element of $(W\ \mathbb{N}_3\ C)$

$$
\begin{array}{l}
(sup\ 2\ [i : \mathbb{N}_2](cases\ i\ of \\
\qquad 0 \Rightarrow (sup\ 1\ [j : \mathbb{N}_1](cases\ j\ of\ 0 \Rightarrow (sup\ 0\ [k : \mathbb{N}_0](cases\ k\ of)))) \\
\quad\ \mid 1 \Rightarrow \\
\qquad\qquad (sup\ 2\ [l : \mathbb{N}_2](cases\ l\ of \\
\qquad\qquad\qquad 0 \Rightarrow (sup\ 0\ [k : \mathbb{N}_0](cases\ k\ of)) \\
\qquad\qquad\quad\ \mid 1 \Rightarrow (sup\ 1\ [j : \mathbb{N}_1](cases\ j\ of \\
\qquad\qquad\qquad\qquad 0 \Rightarrow (sup\ 0\ [k : \mathbb{N}_0](cases\ k\ of))))))))
\end{array}
$$

(Of course, for practical uses we have to define some syntactic tools, to spare the user the pain of writing such terms.)

General Tree Types. To deal with multi-sorted signatures we need to generalize the construction. The General Trees type constructor that we use is very similar to that introduced by Kent Petersson and Dan Synek in [20] (see also [19], chapter 16).

In the multi-sorted case we have to define not just one type of terms, but n types, if n is the number of sorts. These types are mutually inductive. So we define a family $\mathbb{N}_n \to *^s$. In general we consider the case in which we want to define a family of tree types indexed on a given type A, so the elements of A are though of as indexes for the sorts. For what regards the functions, besides their arity we have to take into account from which sort each argument comes and to which sort the result belongs. Like before we have a type B of indexes for the functions. To each $b : B$ we have to associate, as before, a set $(C\ b)$ indexing its arguments. But now we must also specify the type of the arguments: to each $c : (C\ b)$ we must associate a sort index (the sort of the corresponding argument) $(g\ b\ c) : A$. Therefore we need a function $g : (b : B)(C\ b) \to A$. Furthermore we must specify to which sort the result of the application of the function b belongs, so we need an other function $f : B \to A$. Then in the context

$$
\begin{array}{llll}
A, B & : *^s & \qquad & f : B \to A \\
C & : B \to *^s & \qquad & g : (b : B)(C\ b) \to A
\end{array}
$$

we define the inductive family of types $(General\_tree\ A\ B\ C\ f\ g) : A \to *^s$ with the constructor (we write Gt for $(General\_tree\ A\ B\ C\ f\ g)$)

$$g\_tree : (b : B)((c : (C\ b))(Gt\ (g\ b\ c))) \to (Gt\ (f\ b)))$$

In the case of a signature $\sigma = \langle n, [f_1, \ldots, f_m]\rangle$ such that for every $i : \mathbb{N}_m$, $f_i = \langle [a_{i,0}, \ldots, a_{i,k_i-1}], r_i\rangle$ (k_i is the arity of f_i), we have

$$
\begin{array}{lll}
A & = \mathbb{N}_n & \qquad\qquad f = [b : B]r_b \\
B & = \mathbb{N}_m & \qquad\qquad g = [b : B][c : (C\ b)]a_{b,c} \\
(C\ b) & = \mathbb{N}_{k_b} \quad \text{for every } b : B &
\end{array}
$$

The family of types of terms is $(Term\ \sigma) := (General\_tree\ A\ B\ C\ f\ g)$.

The problem of intensionality. One problem that arises when using the General Trees constructor to define term algebras is the intensionality of equality. A term (tree) is defined by giving a constructor $b : B$ and a function

$h : (c : (C\ b))(Term\ \sigma\ (g\ b\ c))$. It is possible that two functions $h_1, h_2 : (c : (C\ b))(Term\ \sigma\ (g\ b\ c))$ are extensionally equal, i.e. $(h_1\ c) = (h_2\ c)$ for all $c : (C\ b)$, but not intensionally equal, i.e. not convertible. In this case the two trees $(g\_tree\ b\ h_1)$ and $(g\_tree\ b\ h_2)$ are intensionally distinct. But we want two terms obtained by applying the same function to equal arguments to be equal. Since algebras are required to be setoids, and not just sets, we can solve this problem by defining an inductive equivalence relation on the types of terms that captures this extensionality,

$$\begin{array}{l} Inductive\ tree\_eq : (a : A)(Term\ \sigma\ a) \to (Term\ \sigma\ a) \to *^p := \\ \quad tree\_eq\_intro : (b : B)(h_1, h_2 : (c : (C\ b))(Term\ \sigma\ (g\ b\ c))) \\ \qquad\qquad ((\forall c : (C\ b))(tree\_eq\ (g\ b\ c)\ (h_1\ c)\ (h_2\ c))) \to \\ \qquad\qquad (tree\_eq\ (f\ b)\ (g\_tree\ b\ h_1)\ (g\_tree\ b\ h_2)) \end{array}$$

Now we would like to prove that $(tree\_eq\ a)$ is an equivalence relation on $(Tree\ \sigma\ a)$. Unfortunately no proof of transitivity could be found. The problem can be formulated and generalized in the following way. If we have an inductive family of types and a generic element of one of the types in the family, it is not generally possible to prove an inversion result stating that the form of the element is an application of one of the constructors corresponding to that type. This was proved for the first time by Hofmann and Streicher in the case of the equality types (see [11]). Therefore we just took the transitive closure of the above relation.

Functions. We have constructed an interpretation of the sorts of a signature in setoids of terms as syntax trees. We still have to interpret the functions. This is not difficult given the way we defined the function interpretation. The functions are associated to the elements of the type $B = \mathbb{N}_m$. Given an element $b : B$ we have a function

$$(g\_tree\ b) : ((c : (C\ b))(Term\ \sigma\ (g\ b\ c))) \to (Term\ \sigma\ (f\ b))$$

It is straightforward to prove that it preserves the setoid equality, so it has the right type to be the interpretation of the function symbol b. Let us call $(functions\_interpretation\ \sigma)$ this family of setoid functions. We then obtain the algebra of terms

$$\begin{array}{ll} Term\_algebra & : (\sigma : Signature)(Algebra\ \sigma) \\ (Term\_algebra\ \sigma) & = (algebra\ \sigma\ (Term\ \sigma)\ (functions\_interpretation\ \sigma)). \end{array}$$

Expressions Algebras. We defined algebras whose elements are closed terms constructed from the function symbols of the signature σ. It is also very important to have algebras of open terms, or expressions, where free variables can appear. To do this we modify the definition of term algebras allowing a set of variables alongside that of functions. So in the construction of syntax trees some leaves may consist of variable occurrences. We assume that every sort has a countably infinite number of variables, so the set of variables is $Var := \mathbb{N}_n \times \mathbb{N}$. Then a variable is a pair $\langle s, n\rangle$ where s determines the sorts to which it belongs and n says that it is the n-th variable of that sort. Variables are treated as constants, i.e. as function symbols of zero arity. In the definition of the term algebra we modify the set B of constructors: $B := \mathbb{N}_m + Var$ and also the family C giving the types of the subtrees in such a way that $(C\ (inr\ v))$ is the empty set for every variable v, while $(C\ (inl\ j)) = \mathbb{N}_{k_j}$ as before. The rest of the definition remains the same. We may also abstract from the actual set of variables and use any family $X : \mathbb{N}_n \to *^s$ as the family of sets of variables. The closed terms are then a particular case obtained by taking $(X\ i) \equiv \emptyset$ for every $i : \mathbb{N}_n$, and the previous case is obtained by taking $(X\ i) \equiv \mathbb{N}$.

5 Congruences, Quotients, Subalgebras, and Homomorphisms

Congruences and quotients. If σ is a signature and $\mathcal{A}$ a σ-algebra, then we call congruence a family of equivalence relations over the sorts of $\mathcal{A}$ that is consistent with the operations of the algebra, i.e. when we apply one of the operations to arguments that are in relation then we obtain results that are in relation. Such condition is rendered in Type Theory by the following

Definition 14. *Let $\sigma \equiv \langle n, [f_0, \ldots, f_{m-1}]\rangle : Signature$ with $f_i \equiv \langle [a_{i,0}, \ldots, a_{i,h_i}], r_i\rangle$ for $i : \mathbb{N}_m$, and $\mathcal{A} : (Algebra\ \sigma)$. A family of relations on the sorts $A = (sorts\ \mathcal{A})$, $(\equiv_s) : (Setoid\_relation\ (A\ s))$ for $s : \mathbb{N}_n$, satisfies the* substitutivity condition *$(Substitutivity\ (\equiv))$ if and only if*

$$\begin{aligned}&(\forall i : \mathbb{N}_m)(\forall args_1, args_2 : (Fun\_arg\_arguments\ \mathcal{A}\ i))\\&((\forall j : \mathbb{N}_{h_i})(args_1\ j) \equiv_{a_{i,j}} (args_2\ j)) \to (f_{i\mathcal{A}}\ args_1) \equiv_{r_i} (f_{i\mathcal{A}}\ args_2).\end{aligned}$$

The type of congruences *over a σ-algebra $\mathcal{A}$ is the record type*

$$\begin{aligned}&Record\ Congruence\ : \Box :=\\&congruence \begin{cases} cong\_relation : (s : \mathbb{N}_n)(Setoid\_relation\ (A\ s))\\ cong\_equiv\quad : (s : \mathbb{N}_n)(Equiv\ (cong\_relation\ s))\\ cong\_subst\quad : (Substitutivity\ cong\_relation)\end{cases}\end{aligned}$$

So a congruence on $\mathcal{A}$ has the form $(congruence\ rel\ eqv\ sbs)$ where rel is a family of setoid relations on the sorts of $\mathcal{A}$, eqv is a proof that every element of the family is an equivalence relation and sbs is a proof that the family satisfies the substitutivity condition.

Given a congruence over an algebra we can construct the quotient algebra. In classic Universal Algebra this is done by taking as sorts the sets of equivalence classes with respect to the congruence. In Type Theory, as we have already said about quotients of setoids, the quotient has exactly the same carriers, but we replace the equality relation. The substitutivity condition guarantees that what we obtain is still an algebra.

Lemma 1. *Let $\sigma : Signature$, $\mathcal{A} : (Algebra\ \sigma)$ and $(\equiv) : (Congruence\ \sigma\ \mathcal{A})$. If we consider the family of setoids obtained by replacing each $=_{(sorts\ \mathcal{A}\ s)}$ by $\equiv_s$ the functions of $\mathcal{A}$ are still well defined. We can therefore define the* quotient algebra *$\mathcal{A}/_\sigma \equiv$.*

Subalgebras. The definition of subalgebra can be given in the same spirit of the definition of quotient algebras.

Definition 15. *Let $\mathcal{A}$: (Algebra σ) and $\mathcal{P}_s$: (Setoid_predicate (sorts $\mathcal{A}$ s)) a family of predicates on the sorts of $\mathcal{A}$. We say that $\mathcal{P}$ is* closed under the functions *of $\mathcal{A}$ if*

$$\begin{array}{l}(\forall i : \mathbb{N}_m)(\forall args : (Fun\_arg\_arguments\ \mathcal{A}\ i))\\((\forall j : \mathbb{N}_{h_i})(\mathcal{P}_{a_j}\ (args\ j))) \to (\mathcal{P}_{r_j}\ (f_{i\mathcal{A}}\ args)).\end{array}$$

Definition 16. *The* subalgebra $\mathcal{A}|_\sigma\mathcal{P}$ *is the σ-algebra with sorts (sorts $\mathcal{A}$ s)$|\mathcal{P}_s$ and functions the restrictions of the functions of $\mathcal{A}$.*

Notice that the restrictions of the functions of $\mathcal{A}$ to the subsetoids (*sorts* $\mathcal{A}$ $s)|\mathcal{P}_s$ are well-defined because $\mathcal{P}$ is closed under function application. The proof of this fact gives the proof of $(\mathcal{P}_{r_j}\ (f_{i\mathcal{A}}\ args))$ and therefore allows the construction of a well-typed element of the Σ-type which is the carrier of the subsetoid.

Homomorphisms. Given a signature σ : *Signature* and two σ-algebras $\mathcal{A}$ and $\mathcal{B}$, we want to define the notion of homomorphism between $\mathcal{A}$ and $\mathcal{B}$. Informally an homomorphism is a family of functions $\phi_s : (A\ s) \to (B\ s)$, where $s : \mathbb{N}_n$ and A and B are the families of sorts of $\mathcal{A}$ and $\mathcal{B}$ respectively, that commutes with the interpretation of the functions of σ. That means that if f is one of the function types of σ and $a_1, \dots, a_k$ are elements of the algebra A, belonging to the sorts prescribed by the types of the arguments of f, then, suppressing the index i in ϕ_i, $(\phi\ (\|f\|_\mathcal{A}\ a_1\ \dots\ a_k)) = (\|f\|_\mathcal{B}\ (\phi\ a_1)\ \dots\ (\phi\ a_k))$ where $\|f\|_\mathcal{A}$ indicates the curried version of the interpretation of the function type f in the algebra $\mathcal{A}$.

Formally we have first of all to require that ϕ is a family of setoid functions $\phi : (i : \mathbb{N}_n)(A\ i)[\to](B\ i)$. Then the requirement that ϕ must commute with the functions of the signature must take into account the way we interpreted the function symbols. Let $i : \mathbb{N}_m$ be a function index, and $f_i = \langle [a_{i,0}, \dots, a_{i,k_i-1}], r_i\rangle$ be the corresponding function type of σ. Assume we have an argument function for $f_{i\mathcal{A}}$, $args_\mathcal{A}$: (*Fun_arg_arguments* $\mathcal{A}$ i). Remember that this is a function that to every $j : \mathbb{N}_{k_i}$ assign an element $(args_\mathcal{A}\ j)$: (*sorts* $\mathcal{A}$ $a_{i,j}$). Then by applying ϕ to each argument we obtain an argument function for $f_{i\mathcal{B}}$, $args_\mathcal{B} := [j : \mathbb{N}_{k_i}](\phi_{a_{i,j}}\ (args_\mathcal{A}\ j)))$: (*Fun_arg_arguments* $\mathcal{B}$ i). For ϕ to be an homomorphism we must then require that for every function index i the equality $(\phi_{r_i}\ (f_{i\mathcal{A}}\ args_\mathcal{A})) =_{(B\ r_i)} (f_{i\mathcal{B}}\ args_\mathcal{B})$ holds. Let us call this property (*Is_homomorphism* ϕ). Then we can define the type of homomorphisms as the record

$$\begin{array}{l}Record\ Homomorphism\ : *^s\ :=\\ \quad homomorphism \begin{cases} hom\_function : (i : \mathbb{N}_n)(A\ i)[\to](B\ i)\\ hom\_proof \quad\ : (Is\_homomorphism\ \phi)\end{cases}\end{array}$$

By requiring that the setoid functions ϕ_i are injective, surjective or bijective we get respectively the notions of monomorphism, epimorphism and isomorphism. We also call endomorphisms (automorphisms) the homomorphisms (isomorphisms) from an algebra A to itself.

Term evaluation. One important homomorphism is the one from the term algebra $\mathcal{T} = (Term\_algebra\ \sigma)$ to any σ-algebra $\mathcal{A}$. This homomorphism is unique since the interpretation of all terms is determined by the interpretation of functions. *term_evaluation* can be defined by induction on the tree structure of terms in such a way that $(term\_evaluation\ (f_{i\mathcal{T}}\ args)) = (f_{i\mathcal{A}}\ args')$ where $args' := [j : \mathbb{N}_{k_i}](term\_evaluation\ (args\ j))$ and we have suppressed the sort indexes. After proving that *term_evaluation* is a setoid function (preserves the equality of terms) and that it commutes with the operations of σ, we obtain an homomorphism $term\_ev : (Homomorphism\ \sigma\ \mathcal{T}\ \mathcal{A})$.

Similarly we can define the evaluation of expressions containing free variables. In this case the function *expression_evaluation* takes an additional argument $ass : (Assignment\ \sigma\ \mathcal{A})$ assigning a value in the right sort of $\mathcal{A}$ to every variable: $(Assignment\ \sigma\ \mathcal{A}) := (v : (Var\ \sigma))(A\ (\pi_1\ v))$. Using this extra argument to evaluate the variables, we can construct as before an homomorphism $expression\_ev : (Homomorphism\ \sigma\ \mathcal{E}\ \mathcal{A})$ where $\mathcal{E} = (Expressions\_algebra\ \sigma)$.

Kernel of a homomorphism. Associated to every homomorphism of σ-algebra $\phi : (Homomorphism\ \sigma\ \mathcal{A}\ \mathcal{B})$ there is a congruence on $\mathcal{A}$ called the *kernel* of ϕ.

Definition 17. *The* kernel *of the homomorphism ϕ is family of relations*

$$\begin{array}{ll}(ker\_rel\ \phi) & : (s : \mathbb{N})(relation\ (A\ s)) \\ (ker\_rel\ \phi\ s\ a_1\ a_2) & \Longleftrightarrow\ (\phi_s\ a_1) =_{(B\ s)} (\phi_s\ a_2)\end{array}$$

Lemma 2. *ker_rel is a congruence on $\mathcal{A}$.*

The kernel of ϕ is indicated by the standard notation $\equiv^\phi$.

We can, therefore, take the quotient $\mathcal{A}/_\sigma \equiv^\phi$ and consider the natural homomorphism between $\mathcal{A}$ and $\mathcal{A}/_\sigma \equiv^\phi$. In classic Universal Algebra this homomorphism associates to every element a in $\mathcal{A}$ the equivalence class $[a]_{\equiv^\phi}$. But in our implementation the carriers of $\mathcal{A}$ and $\mathcal{A}/_\sigma \equiv^\phi$ are the same and so the natural homomorphism is just the identity. We only have to verify that it is actually an homomorphism (it preserves the setoid equality and it commutes with the operation of the signature).

Lemma 3. *For any σ-algebra $\mathcal{A}$ and any congruence $\equiv$ on $\mathcal{A}$, the family of identity functions $[s : \mathbb{N}_n][x : (sorts\ \sigma\ \mathcal{A}\ s)]x$ is a homomorphism from $\mathcal{A}$ to $\mathcal{A}/_\sigma \equiv$.*

In the case of $\equiv^\phi$ such homomorphism is indicated by nat^ϕ.

First homomorphism theorem. Once we have developed the fundamental notions of Universal Algebra in Type Theory and we have constructed operators to manipulate them, we can prove some standard basic results like the following.

Theorem 1 (First Homomorphism Theorem). *Let $\mathcal{A}$ and $\mathcal{B}$ be two σ-algebras and ϕ : (Epimorphism σ $\mathcal{A}$ $\mathcal{B}$). Then there exists an isomorphism*

$$(\mathit{ker\_quot\_iso}\ \phi) : (\mathit{Isomorphism}\ \sigma\ \mathcal{A}/_{\sigma} \equiv^{\phi}\ \mathcal{B})$$

such that $(\mathit{ker\_quot\_iso}\ \phi) \circ \mathit{nat}^{\phi} = \phi$, where the equality is the extensional functional equality.

6 Conclusions and Further Research

We have implemented in Type Theory (using the proof development system Coq for the formalization) the fundamental notions and results of Universal Algebras. This implementation allows us to specify any first order algebraic structure and has operators to construct free algebras over a signature. We defined the constructions of subalgebras, product algebras and quotient algebras and proved their basic properties. There were two main points in which we had to employ special type theoretic constructions: we used setoids as carriers for algebras in order to be able to define quotient algebras and we used wellorderings to represent free algebras.

This implementation is intended to serve two purposes. From the practical point of view it provides a set of tools that make the use of Type Theory in the development of mathematical structures easier. From the theoretical point of view it investigates the use of Type Theory as a foundation for Mathematics.

An important line of research that we intend to pursue is that of equational reasoning. The next steps in this direction are the definition of equational classes of algebras, where equations are represented by pairs of open terms, and the proof of Birkhoff's soundness theorem. This will give us tools to automatically prove formulas over a generic algebra by lifting them to the syntactic level of expressions.

References

[1] Peter Aczel. Notes towards a formalisation of constructive galois theory. draft report, 1994.

[2] H. P. Barendregt. Lambda Calculi with Types. In S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 2.* Oxford University Press, 1992.

[3] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Yann Coscoy, David Delahaye, Daniel de Rauglaudre, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, Henri Laulhère, César Muñoz, Chetan Murthy, Catherine Parent-Vigouroux, Patrick Loiseleur, Christine Paulin-Mohring, Amokrane Saïbi, and Benjanin Werner. *The Coq Proof Assistant Reference Manual. Version 6.2.*

[4] G. Barthe, M. Ruys, and H. P. Barendregt. A two-level approach towards lean proof-checking. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs (TYPES'95)*, volume LNCS 1158, pages 16–35. Springer, 1995.

[5] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In Martín Abadi and Takayasu Ito, editors, *Theoretical Aspects of Computer Software. Third International Symposium, TACS'97*, volume LNCS 1281, pages 515–529. Springer, 1997.

[6] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[7] Thierry Coquand and Christine Paulin. Inductively defined types. In P. Martin-Löf, editor, *Proceedings of Colog '88*, volume 417 of *Lecture Notes in Computer Science.* Springer-Verlag, 1990.

[8] Thierry Coquand and Henrik Persson. Integrated Development of Algebra in Type Theory. Presented at the Calculemus and Types '98 workshop, 1998.

[9] Eduardo Giménez. A Tutorial on Recursive Types in Coq. Technical report, Unité de recherche INRIA Rocquencourt, 1998.

[10] Martin Hofmann. Elimination of extensionality in Martin-Löf type theory. In Barendregt and Nipkow, editors, *Types for Proofs and Programs. International Workshop TYPES '93*, pages 166–190. Springer-Verlag, 1993.

[11] Martin Hofmann and Thomas Streicher. A groupoid model refutes uniqueness of identity proofs. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 208–212. IEEE Computer Society Press, 1994.

[12] Douglas J. Howe. Computational metatheory in Nuprl. In E.Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, volume LNCS 310, pages 238–257. Springer-Verlag, 1988.

[13] Gérard Huet and Amokrane Saïbi. Constructive category theory. In *In honor of Robin Milner.* Cambridge University Press, 1997.

[14] Paul Jackson. Exploring abstract algebra in constructive type theory. In *Automated Deduction – CADE-12*, volume Lectures Notes in Artificial Intelligence 814, pages 591–604. Springer-Verlag, 1994.

[15] Zhaohui Luo. *Computation and Reasoning, A Type Theory for Computer Science*, volume 11 of *International Series of Monographs on Computer Science.* Oxford University Press, 1994.

[16] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science, VI, 1979*, pages 153–175. North-Holland, 1982.

[17] Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984. Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980.

[18] K. Meinke and J. V. Tucker. Universal Algebra. In S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 1.* Oxford University Press, 1992.

[19] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory.* Clarendon Press, 1990.

[20] Kent Petersson and Dan Synek. A Set Constructor for Inductive Sets in Martin-Löf's Type Theory. In *Proceedings of the 1989 Conference on Category Theory and Computer Science, Manchester, U.K.*, volume 389 of *Lecture Notes in Computer Science.* Springer-Verlag, 1989.

[21] Amokrane Saïbi. Typing algorithm in type theory with inheritance. In *POPL'97: The 12th ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages*, pages 292–301. Association for Computing Machinery, 1997.

[22] Milena Stefanova. *Properties of Typing Systems.* PhD thesis, Computer Science Institute, University of Nijmegen, 1999.

[23] Laurent Théry. Proving and Computing: a certified version of the Buchberger's algorithm. Technical report, INRIA, 1997.