Initial Semantics for higher-order typed syntax

Benedikt Ahrens
University of Nice – Sophia Antipolis, France
ahrens@unice.fr

Julianna Zsidó
Université Montpellier II, France
jzsido@univ-montp2.fr

December 7, 2010

Abstract

We present an initial semantics result for typed higher-order syntax based on monads and modules over monads. The notion of module generalizes the substitution structure of monads.

For a simply typed binding signature $S$ we define a representation of $S$ to be a monad equipped with a morphism of modules for each of its arities. The monad of abstract syntax of $S$ then is the initial object in the category of representations of $S$, and the constructors are module morphisms.

Our theory is implemented and proved correct in the proof assistant Coq, making heavy use of dependent types. The monadic theory gives rise to an implementation of syntax where both terms and variables are intrinsically typed. Terms are implemented as doubly parametrized data types, dependent on a typed set of variables as well as on object types.

The implementation is fully constructive. Given any signature, the implementation automatically provides the initial representation, recursion principles, a certified substitution operation and – via initiality – a morphism towards any other representation of $S$.

Introduction

We present an initial semantics result for typed higher-order syntax based on monads and modules over monads. Our theory is not the first of its kind, cf. the part about related work. It is, however, the only one which features monads and modules and is fully implemented in a proof assistant.

Altenkirch and Reus [1] recognized monads as a convenient categorical device to talk about substitution. They characterize the untyped lambda calculus as a monad over the category of sets.

In order to account for types, our basic category of interest is the category $[T, \text{Set}]$ of families of sets indexed by a set $T$ of object types, the objects of which we will also refer to by “typed sets” [1]. Our monads are monads over $[T, \text{Set}]$.

1It is, however, important to notice that we are not talking about objects of the slice category $\text{Set} / T$. 
The notion of module over a monad \[18\] generalizes monadic substitution to functors that are not necessarily endofunctors. A module is hence a functor equipped with a substitution map. Morphisms of modules are natural transformations which are compatible with the module substitution.

We characterize the syntax associated to a signature \(S\) as an initial object in the category of so-called representations of \(S\). An object of this category is a monad over typed sets equipped with a morphism of modules for each arity. A morphism of representations is a morphism between the underlying monads which is compatible with the morphisms of modules. For the initial representation these module morphisms are given by the constructors of the syntax, and the property of being a module morphism captures their commutativity with substitution.

Our theory is implemented in the proof assistant Coq \[8\]. This implementation can be seen as a formal proof of a mathematical theory in a constructive setting, and as such delivers confidence in the correctness of the theorem.

Perhaps more importantly, the theory translates to an implementation of syntax using exclusively intrinsic typing, a style of implementation that has been advertised by Benton et al. \[3\]. Here typing is not done by a typing judgement, given by, say, an inductive predicate. Instead it is referred to the typing of the meta-language. The technique and its benefits are discussed in \[3\].

The monadic view (in contrast to the operadic view taken by others) allows for a further improvement in this direction compared to Benton et al. By characterizing abstract syntax as a monad over typed sets we extend intrinsic typing to variables, i.e. the object type of each variable is given intrinsically by a type parameter of the meta-language. Accordingly there is no more need for an explicit typing device for variables such as Benton et al.’s environments.

Related work

The theory we present was first published in Zsidó’s PhD thesis \[31\]. It is based on the work by Hirschowitz and Maggesi on the untyped lambda calculus \[19\].

Initial semantics For untyped first-order syntax the notion of initial algebra has been coined by Goguen et al. \[16\] in the 1970s. Semantics for syntax with binding were presented by several people independently. Gabbay and Pitts’ nominal approach \[14,15,27\] uses a set theory enriched with atoms to establish an initiality result. Fiore, Plotkin and Turi \[13,10,11\] modify the classical theory of initial algebras by replacing sets by presheaves over finite sets, constructing fresh, to-be-bound variables by a \(+1\) functor, corresponding to context extension, on those presheaves. Hofmann \[21\] and Miculan and Scagnetto \[25\] present variable binding in a Higher-Order Abstract Syntax (HOAS) style. Later Power and Tanaka generalize and subsume those three theories to a general category of contexts \[30\]. An informal overview of this work and references to more technical papers can be found in \[28\].

The very general picture of these theories is in fact quite similar to the first initial algebra semantics, even if technically much more involved: for a given signature, a signature endofunctor \(\Sigma\) over a suitable category is defined. The objects of interest are then the \(\Sigma\)-algebras that are compatible with some substitution structure.
In her PhD thesis [31] the second author closely examines the relation between our theory and that of Fiore et al.. She establishes an adjunction between our category of representations and that of Σ–monoids, its unit being an isomorphism. In the untyped case she additionally describes the very close relation of the signature functor Σ and the codomain modules by means of mates under certain adjunctions. This relation, however, does not generalize to the typed case in the same way.

The first author is working on extending the presented result in several directions. Firstly, the category of representations can be enlarged to allow for representations of a T-signature in a monad over [U, Set] for a given “translation of object types” \( f : T \to U \). In this way the compilation from one programming language to another (over different object types) can be seen as a morphism of representations. Secondly, syntax can be enriched with a reduction relation. The computer-implemented example of the compilation from PCF to the untyped lambda calculus, each equipped with the usual reduction relation, can be found on the first author’s web page [2].

**Implementation of syntax** The implementation and formalization of syntax has been studied by a variety of people. The POPLMARK challenge [2] is a benchmark which aims to evaluate readability and provability when using different techniques of variable binding. The technique we used, called Nested Abstract Syntax, is used in a partial solution [20], but was proposed earlier by others, e. g. [5] [1]. The use of intrinsic typing by dependent types of the meta–language was advertised, among others, in [5].

During our work we became aware of Capretta and Felty’s framework for reasoning about programming languages [6].

They implement a tool – also in the Coq proof assistant – that, given a signature, provides the associated abstract syntax as a data type dependent on the object types. Variables are encoded using de Bruijn indices. They then add a layer to translate those terms into syntax using named abstraction, and provide suitable induction and recursion principles. Their tool may hence serve as a practical framework for reasoning about programming languages. Our implementation remains on the theoretical side by not providing named syntax and exhibiting the category theoretical properties of abstract syntax.

**Synopsis**

The first and second sections are concerned with the theory while the third is devoted to the implementation and its use as a framework for language design.

In the first section, we define the notion of monad (as kleisli structure) and (left) module over a monad as well as their resp. morphisms. Those familiar with these notions may skip this section or just use it as a reference for our notations. Some examples of monads and modules are given, which will later be of importance.

The second section introduces the notion of arity, signature and representations of signatures in suitable monads. We then state our main result: the category of representations has an initial object.

Throughout the first two sections the names in typewriter font denote the names of the objects as they are in the computer implementation.

The implementation of the theory in the proof assistant Coq is explained in the third section. At first we give a (very brief) introduction to Coq. We proceed by presenting the implementation of the notions introduced in the previous sections. Afterwards an explicit definition of the initial object is given (cf. [18]).

Some conclusions and future work are stated in the last section.

1 Monads & Modules

Monads have long been known to capture the notion of substitution, cf. [11]. The closely connected notion of module over a monad, however, was only recently popularized by Hirschowitz and Maggesi [18]. Similarly to the two equivalent definitions of monads as presented by Manes [24] there are two equivalent definitions of modules over a monad. Contrary to the given reference [18] we present our theory using the “kleisli” version of monads and modules.

1.1 Definitions

Definition 1 (Monad): A monad \( P \) over a category \( C \) is given by

- a map \( P : C \to C \) on the objects of \( C \) (observe the abuse of notation),
- for each object \( c \) of \( C \), a morphism \( \eta_c \in C(c, Pc) \) and
- for all objects \( c \) and \( d \) of \( C \) a substitution map

\[
\sigma_{c,d} : C(c, Pd) \to C(Pc, Pd)
\]

such that the following diagrams commute for all suitable morphisms \( f \) and \( g \).

![Diagram](image)

The name “substitution map” is justified when e. g. \( C \) is instantiated with the category of sets and \( PX \) is a set of terms with variables in the set \( X \). The diagrams then express the well-known substitution properties, cf. [11]. We refer to these properties as fusion properties.

A monad \( P \) can be equipped with a functorial structure by setting

\[
P(f) := \text{lift}_P(f) := \sigma(\eta \circ f),
\]
the functoriality axioms being a consequence of the monad axioms (lift).

For two monads $P$ and $Q$ over the same category $C$ a morphism of monads is a family of morphisms $\tau_c \in C(Pc, Qc)$ that is compatible with the monadic structure.

**Definition 2 (Monad Hom):** A morphism of monads from $P$ to $Q$ is given by a collection of morphisms $\tau_c \in C(Pc, Qc)$ such that the following diagrams commute for all suitable morphisms $f$.

\[
\begin{array}{ccc}
P_c & \xrightarrow{\sigma^P(f)} & Pd \\
\downarrow \tau_c & & \downarrow \tau_d \\
Qc & \xrightarrow{\sigma^Q(\tau_c \circ f)} & Qd
\end{array}
\]

As a consequence from these commutativity properties the family $\tau$ is a natural transformation between the functors induced by the monads $P$ and $Q$.

Given a monad $P$ over $C$, the notion of module over $P$ generalizes the notion of monadic substitution.

**Definition 3 (Module):** Let $D$ be a category. A module $M$ over $P$ with codomain $D$ is given by

- a map $M : C \to D$ on the objects of the categories involved and
- for all objects $c, d$ of $C$ a map

  $\varsigma_{c,d} : C(c, Pd) \to C(Mc, Md)$

such that the following diagrams commute for all suitable morphisms $f$ and $g$:

\[
\begin{array}{ccc}
Mc & \xrightarrow{\varsigma(f)} & Md \\
\downarrow \varsigma(\sigma(g) \circ f) & & \downarrow \varsigma(g) \\
Me & & Me
\end{array}
\]  

\[
\begin{array}{ccc}
Mc & \xrightarrow{\varsigma(g)} & Me \\
\downarrow \varsigma(\eta_c) & & \downarrow \text{id} \\
Mc & & Me
\end{array}
\]

A functoriality for such a module $M$ can then be defined in the same way as for monads (mlift).

A module morphism is a family of morphisms that is compatible with module substitution.

**Definition 4 (Mod Hom):** Let $M$ and $N$ be two modules over $P$ with codomain $D$. A morphism of $P$–modules from $M$ to $N$ is given by a collection of morphisms $\rho_c \in D(Mc, Nc)$ such that for all morphisms $f \in C(c, Pd)$ the following diagram commutes:

\[
\begin{array}{ccc}
Mc & \xrightarrow{\varsigma^M(f)} & Md \\
\downarrow \rho_c & & \downarrow \rho_d \\
Nc & \xrightarrow{\varsigma^N(f)} & Nd.
\end{array}
\]
A module morphism $M \rightarrow N$ also is a natural transformation $M \rightarrow N$ between the induced functors.

The modules on $P$ with codomain $D$ and morphisms between them form a category called $\text{Mod}_P^D$ ($\text{MOD} \ P \ D$).

1.2 Examples

The following examples seem rather trivial, but they play a central role in our theory.

Tautological Module ($\text{Taut}_\text{Mod}$): Every monad $P$ over $C$ can be viewed as a module (also denoted by $P$) over itself, i.e. as an object in the category $\text{Mod}_P^C$.

Constant and terminal module ($\text{Const}_\text{Mod}$, $\text{MOD}_{\text{Terminal}}$): For any object $d \in D$ the constant map $T_d: C \rightarrow D$, $c \mapsto d$ for all $c \in C$ can be provided with the structure of a $P$-module for any monad $P$. In particular, if $D$ has a terminal object $1_D$, then the constant module $c \mapsto 1_D$ is terminal in $\text{Mod}_P^D$.

Pullback module ($\text{PbMod}$): Given $h$ a morphism of monads $P \rightarrow Q$ and $M$ a $Q$-module with codomain $D$, we can define a $P$-module $h^*M$ by setting

$$\varsigma_{h^*M}(f) := \varsigma^M(h_d \circ f).$$

This module is called the pullback module of $M$ along $h$. The pullback can be extended to module morphisms and is functorial.

Induced module morphism ($\text{PbMod}_{\text{ind Hom}}$): With the same notation as in the previous example, the monad morphism $h$ induces a morphism of $P$-modules $h: P \rightarrow h^*Q$.

Products ($\text{Prod}_\text{Mod}$): Suppose the category $D$ is equipped with a product. Let $M$ and $N$ be $P$-modules with codomain $D$. Then the map

$$C \rightarrow D, \ c \mapsto Mc \times Nc$$

can be extended to a module called the product of $M$ and $N$. This construction can again be extended to a product on $\text{Mod}_P^D$.

The following two constructions – derivation and fibre – apply to monads and modules over the category of (families of) sets.

1.3 Derivation

Roughly speaking, a binding constructor makes free variables disappear. Its input are hence terms “with (one or more) additional free variables” compared to the output. Derivation is the process of producing those then-to-be-bound additional variables.

Formally, given a set $V$ ($V$ for variables), we consider a new set

$$V^* := V \amalg \{\ast\}$$

which denotes $V$ enriched with a new distinguished element (the “fresh” variable). The map $V \mapsto V^*$ can be extended to a monad on the category of sets and is hence functorial.
Let $T$ be a discrete category (a set). For each element $u$ of $T$ an object $V$ of the functor category $[T, \text{Set}]$ can be enriched with respect to $u$ by setting

$$V^u(t) := \begin{cases} V(t)^*, & \text{if } t = u \\ V(t) & \text{otherwise,} \end{cases}$$

i.e. we add a fresh variable of type $u$. This yields a monad $(\cdot)^u$ on $[T, \text{Set}]$ (opt\_monad $u$).

Given a monad $P$ over $[T, \text{Set}]$ and a $P$–module $M$ with codomain $[T, \text{Set}]$, we define the derived module w.r.t. $u \in T$ by setting

$$\partial_u M(V) := M(V^u),$$

that is, by forgetting all but one component of the indexed family of sets. This yields a module $M_u \in \text{Mod}^P_{\text{Set}}$. The construction extends to a functor $(\text{ITFIB}\_\text{MOD} \ u)$

The pullback operation commutes with products, derivations and fibres in the following sense.

**Lemma 5:** Let $\mathcal{C}$ be a category and $\mathcal{D}$ be a category with products. Let $P$ and $Q$ be modules over $\mathcal{C}$ and $\rho : P \to Q$ a monad morphism. Let $M$ and $N$ be $P$–modules with codomain $\mathcal{D}$. Then we can construct an isomorphism of modules

$$\rho^* (M \times N) \to \rho^* M \times \rho^* N.$$  

**Lemma 6:** Consider the setting as in the preceding lemma, with $\mathcal{C} = [T, \text{Set}]$ and $\mathcal{D} = \text{Set}$. Let $u$ be an element of $T$. Then we can construct isomorphisms of modules

$$\rho^* (\partial_u M) \to \partial_u (\rho^* M)$$

and

$$\rho^* (M_u) \to (\rho^* M)_u.$$
2 Signatures & Representations

An *arity* determines the type and binding behaviour of a *constructor*. A *signature* is a family of arities. A *representation* of a signature $S$ is any monad (over a suitable category) accompanied by a module morphism for each arity $\alpha$ of $S$, where the source and target module of this morphism depend on $\alpha$.

Among those representations the object of interest is the *initial* one, i.e. the representation from which there exists exactly one *morphism of representations* to any other representation. The initial representation is called *syntax* associated to $S$, whereas the other representations (with their respective morphisms from the syntax) are the *semantics* of $S$.

For the formal definitions let us fix a set $T$ of object language types.

**Definition 7:** A $T$–arity is a family of types consisting of $t_i \in T$ for $i = 0, \ldots, n$ and $t_{i,j} \in T$ for all $j = 1, \ldots, m_i$ and all $i = 1, \ldots, n$, written

$$(t_1,1 \ldots t_1,m_1) \ldots (t_n,1 \ldots t_n,m_n) t_0 \rightarrow t_0$$

or shorter

$$(\bar{s}_1)t_1, \ldots, (\bar{s}_n)t_n \rightarrow t_0$$

where $\bar{s}_k$ denotes the list of types $t_{k,1} \ldots t_{k,m_k}$.

Intuitively this stands for an operator that binds $m_k$ variables of types $t_{k,1}, \ldots, t_{k,m_k}$ in its $k$–th argument of type $t_k$, and which yields a term of type $t_0$.

**Example 8:** Let $T$ be the inductive set of types defined by

$$T ::= B \mid T \Rightarrow T$$

where $B$ is a set of base types. The signature of the simply typed lambda calculus is given by

$$S = \{\text{abs}_{s,t}, \text{app}_{s,t}\}_{s,t \in T}$$

where

$$\text{abs}_{s,t} := (s)t \rightarrow s \Rightarrow t$$

and

$$\text{app}_{s,t} := s \Rightarrow t, s \rightarrow t.$$
Definition 10: A representation of a $T$–signature $S$ is a pair $(R, r)$, where $r = (r_\alpha)_{\alpha \in S}$, consisting of a monad $R$ and a representation $r_\alpha$ of each arity $\alpha$ of $S$ in $R$.

Morphisms of representations are monad morphisms that are compatible with the representational structure.

Definition 11: Let $(P, p)$ and $(Q, q)$ be representations of a $T$–signature $S$. A morphism of representations $f: (P, p) \to (Q, q)$ is a morphism of monads $f: P \to Q$ such that the following diagram commutes for all arities $\alpha = (\vec{s}_1)_{t_1}, \ldots, (\vec{s}_n)_{t_n} \to t_0$ of $S$.

\[
\begin{array}{ccc}
\prod_{i=1}^{n} (\partial_{\vec{s}_i} P)_{t_i} & \xrightarrow{p_\alpha} & P_{t_0} \\
\prod_{i=1}^{n} (\partial_{\vec{s}_i} f)_{t_i} \downarrow & & \downarrow f_{t_0} \\
f^* \prod_{i=1}^{n} (\partial_{\vec{s}_i} Q)_{t_i} & \xrightarrow{f^* q_\alpha} & f^* Q_{t_0}
\end{array}
\]

To make sense of this diagram it is necessary to recall the examples of modules of section 1.2. The diagram lives in the category Mod$_P$Set. The vertical morphisms are module morphisms induced by the monad morphism $f$, to which functoriality of derivation, fibre and products are applied. Furthermore instances of lemmas 5 and 6 are hidden in the lower left corner. The lower horizontal morphism makes use of the functoriality of the pullback operation, and in the lower right corner we again use the fact that pullback commutes with fibres.

Example 12: The signature of the simply typed lambda calculus is given by the signature in the example 8. A representation in a monad $R$ on $[T, \text{Set}]$ is given by two families of $R$–module morphisms

\[
\text{app}_{s,t}: R_{s \Rightarrow t} \times R_s \to R_t
\]

and

\[
\text{abs}_{s,t}: (\partial_s R)_t \to R_{s \Rightarrow t}.
\]

Notation 13: Representations of a $T$–signature $S$ and morphisms of representations form a category which we call Rep$(S)$.

The main result is stated in the following theorem.

Theorem 14: Let $S$ be a $T$–signature. Then the category Rep$(S)$ has an initial object.

The proof is straightforward. After the construction of the set of syntax trees one shows that it satisfies all the required axioms. For details see Zsidó’s thesis 31.
Example 15: The initial object of the category of representations of the signature of example \( \mathcal{S} \) is the simply typed lambda calculus. Thus we have characterized the simply typed lambda calculus \( \text{STLC}_T \) as a monad on \([T, \text{Set}]\) equipped with two collections of \( \text{STLC}_T \)-module morphisms

\[
\text{app}_{s,t} : (\text{STLC}_T)_s \times (\text{STLC}_T)_s \to (\text{STLC}_T)_t
\]

and

\[
\text{abs}_{s,t} : (\partial_s \text{STLC}_T)_t \to (\text{STLC}_T)_s \Rightarrow t
\]

which are precisely the application and abstraction constructors.

3 Implementation details

We implemented the theorem in the proof assistant Coq. Starting from an arbitrary type of object types \( T \) and a typed binding signature \( S \) over \( T \) we define the category of representations of \( S \) and its initial object. Abstracting over \( S \) and \( T \) then gives a parametrized theory of representations. We obtain a function that, when fed with any signature \( S' \), returns a category of representations and its initial monad.

The implementation makes heavy use of \textit{intrinsic typing} by Coq’s dependent types. The objects of the functor category \([T, \text{Set}]\) are simply families of types indexed by \( T \), so that we do not need any typing judgments to choose the terms and variables with the correct type. Indeed, the complete object language typing is transferred to Coq’s type system. In particular we do not need decidable equality on the type \( T \) of object types.

The implementation is divided into two parts. We have a library of general category theoretical concepts, containing (among others) the definitions and lemmas about monads and modules of section 1. The theory–specific part about signatures and their representations, corresponding to section 2, is separated from the library. Its size is about 1350 lines of code, consisting of 450 lines of specification and 900 lines of proof. The Coq files are available on the first author’s website.

3.1 About the proof assistant Coq

The proof assistant Coq is an implementation of the Calculus of Inductive Constructions (CIC) which itself is a constructive type theory. Its objects are terms built according to a grammar (see the Coq manual for the term forming rules). Each valid term has its associated type which is itself a term and which is automatically computed by Coq. With certain limitations this typing relation is comparable to a set membership relation. In Coq a typing judgment is written \( t : T \), meaning that \( t \) is a term of type \( T \). Typing judgments are for example \( 1 : \text{Nat} \) and \( \text{plus} : \text{Nat} \to \text{Nat} \to \text{Nat} \). Function application is simply denoted by a blank, i.e. we write \( f \ x \) for \( f(x) \).

The CIC also treats propositions as types via the Curry–Howard isomorphism, hence a proof of a proposition \( P \) is in fact a term of type \( P \). Accordingly, a proof of a proposition \( A \Rightarrow B \) is a function \( A \to B \), i.e. a term which associates a proof of \( B \) to any proof of \( A \). As an example, the function

\[ f : \text{Nat} \to \text{Nat} \Rightarrow \text{Nat} \]

is defined as

\[ f(n) = n + 1 \]

for any natural number \( n \).
$P \rightarrow P, x \mapsto x$ is a proof of the tautology $P \Rightarrow P$. In the proof assistant Coq a user hence proves a proposition $P$ by providing a term $p$ of type $P$. Coq checks the validity of the proof $p$ by verifying whether $p : P$.

Coq comes with extensive support to interactively build the proof terms of a given proposition. In proof mode so-called tactics help the user to reduce the proposition they want to prove – the goal – into one or more simpler subgoals, until reaching trivial subgoals which can be solved directly.

### 3.2 Library of category theory

When we started our work on formalizing the initiality theorem, the standard library of Coq was still lacking a usable library of category theory. The best library then available, ConCaT [22], uses a custom implementation of setoids that is a source of universe clashes. We hence implemented a fragment of category theory ourselves. Only later we learned about the efforts of Spitters and v. d. Weegen [29], which hopefully will result in a full–grown library one day.

We implemented functors, natural transformations, monads, (co)products, limits and other notions. However, for our purposes the concepts we present in the following will be sufficient. In some rare cases we mangle the presentation of the source code in order to make the formalization more accessible.

#### Category

The type class of categories is parametrized by a type of objects and a dependent type of morphisms, whose parameters are the source and target objects.

```coq
Class Cat (obj:Type)(mor: obj -> obj -> Type) := {
  mor_oid:> forall a b, Setoid (mor a b);
  id: forall a, mor a a;
  comp: forall {a b c},
    mor a b -> mor b c -> mor a c;
  comp_oid:> forall a b c,
    Proper (equiv ==> equiv ==> equiv)
      (@comp a b c);
  id_r: forall a b (f: mor a b),
    comp f (id b) == f;
  id_l: forall a b (f: mor a b),
    comp (id a) f == f;
  assoc: forall a b c d (f: mor a b)
    (g:mor b c) (h: mor c d),
    comp (comp f g) h == comp f (comp g h) }.
```

In a category each type $\text{mor} \ a \ b$ of morphisms from $a$ to $b$ is equipped with a custom equality. Defining a specific category hence involves defining what equality on their morphisms shall be. Any equivalence relation – in Coq called $\text{Setoid}$ – on $\text{mor} \ a \ b$ can be declared as such. This equality is denoted by the infix symbol "$==". In the following we write "$a \rightarrow b$" for $\text{mor} \ a \ b$.

The composition is a dependent function taking five arguments. Its first three arguments are the objects which are sources and targets of the morphisms to compose. The morphisms themselves are given as fourth and fifth arguments.
The fact that the source of a composite morphism is the same as the source of the first morphism (and similar for the target) is then already determined by the type of the composition function. Furthermore we ask the composition to be a morphism of setoids, i.e., composition should be compatible with the setoidal structure on the morphisms. In the following we will write \( f ;; g \) for the composition of morphisms \( f : a \rightarrow b \) and \( g : b \rightarrow c \).

The basic category of interest \([T, \text{Set}]\) is formalized as a category where objects are collections of types (which now play the role of sets) indexed by \( T \). This category is called \( \text{ITYPE} \ T \).

**Monad** A monad is a type class parametrized by a category \( C \) on objects \( \text{obj} \) and morphisms \( \text{mor} \) and a function \( F : C \rightarrow C \).

```
Class Monad ... (F : C -> C) := {
  weta : forall c, c ---> (F c);
  kleisli : forall a b, 
    (a ---> F b) -> (F a ---> F b);
  kleisli_oid -> forall a b, 
    Proper (equiv ==> equiv) 
    (kleisli (a:=a) (b:=b));
  eta_kl : forall a b (f : a ---> F b), 
    weta a ;; kleisli f == f;
  kl_eta : forall a, kleisli (weta a) == id _;
  dist : forall a b c (f : a ---> F b) 
    (g : b ---> F c), 
    kleisli f ;; kleisli g == 
    kleisli (f ;; kleisli g)
}.
```

The operator \( \text{kleisli} \) is again asked to be a morphism of setoids, similarly to the composition of morphisms.

**Module** A module is a type class which is parametrized by two categories \( C \) and \( D \), a monad \( P \) over \( C \) and a function \( M : C \rightarrow D \) from the objects of \( C \) to the objects of \( D \). It is very similar to the implementation of monads.

```
Class Module (M : C -> D) := {
  mkleisli: forall c d, 
    (c ---> P d) -> (M c ---> M d);
  mkleisli_oid -> forall c d, 
    Proper (equiv ==> equiv) 
    (mkleisli (c:=c)(d:=d));
  mkl_weta: forall c, mkleisli (weta a) == id _;
  mkl_mkl: forall c d e (f : c ---> P d) 
    (g : d ---> P e), 
    mkleisli f ;; mkleisli g == 
    mkleisli (f ;; mkleisli g)
}.
```

*Coq deduces and inserts the missing “object” arguments \( a \), \( b \) and \( c \) automatically from the type of the morphisms. For this reason they are called implicit arguments.*
3.3 Implementation of signatures & representations

A simply typed arity over a type $T$ is just a pair consisting of a list of data for the arguments and an element of $T$ - the type of the output. Each data again is a pair made of a list over $T$ holding the types of the variables which are to be bound in this argument and an element of $T$ – the type of the argument. Hence an arity is a term of type $[[T] * T] * T$, where $[T]$ is custom notation for lists over $T$.

A signature over $T$ as defined in the previous section could be directly implemented as a pair consisting of a type $\text{sig\_index}$ – which is used for indexing the arities – and a map from the indexing type to the actual arity type.

\[
\text{Record Signature : Type := } \{ \\
\text{ sig\_index : Type; } \\
\text{ sig : sig\_index -> } [[T] * T] * T
\}.
\]

A slight modification however turns out to be useful. During the construction of the initial representation a universal quantification over arities with a given target type is needed. We decide to define a signature to be a function which maps each $t : T$ to the set of arities whose output type is the given $t$. In other words, the parameter of $\text{Signature\_t}$ replaces the second component of the arities.

\[
\text{Record Signature\_t (t : T) : Type := } \{ \\
\text{ sig\_index : Type; } \\
\text{ sig : sig\_index -> } [[T] * T]
\}.
\]

\[
\text{Definition Signature := forall t, Signature\_t t}.
\]

As an example we discuss the signature of the simply typed lambda calculus with types $T$ and a type constructor

\[
\rightarrow : T \rightarrow T \rightarrow T.
\]

This constructor corresponds to the arrow constructor $\Rightarrow$ of example 8. We use a different notation since the arrow “$\Rightarrow$” is a built-in Coq notation. At first we define for each $t : T$ an indexing type $\text{TLC\_index\_t}$. After that, we build an indexed signature $\text{TLC\_sig}$ mapping each index to its arity.

\[
\text{Inductive TLC\_index : T -> Type := } \\
\text{ | TLC\_abs : forall s t : T, TLC\_index\_s (s \rightarrow\rightarrow t) } \\
\text{ | TLC\_app : forall s t : T, TLC\_index\_s}
\]

\[
\text{Definition TLC\_sig t := Build\_Signature\_t t}
\]

\[
\text{ (fun r : TLC\_index t => match r with} \\
\text{ | TLC\_abs s t => (s::nil,t)::nil } \\
\text{ | TLC\_app s t => (nil,s \rightarrow\rightarrow t)::(nil,s)::nil}
\text{ end).}
\]

Another example is the signature of $\text{PCF}$, which can be found in our Coq theory files.
Derivation  The addition of a variable (of type \( u \in T \)) to a set of variables \( V \in \mathbb{T}.\text{Set} \) is implemented by a (typed) option datatype. In the given code we add a new variable of object language type \( u : T \) to a set \( V \) of typed variables.

\[
\text{Inductive opt (u : T) (V : \text{ITYPE T}) : \text{ITYPE T} :=}
\]
\[
| \text{some : forall t : T, V t} \rightarrow \text{opt u V t} \\
| \text{none : opt u V u.}
\]

Given a list \( l \) over \( T \), the multiple addition of variables with (object language) types according to \( l \) to a set of variables \( V \) is defined by recursion over \( l \). For this enriched set of variables we introduce the notation \( V ** l \).

\[
\text{Fixpoint pow (l : \mathbb{T}) (V : \text{ITYPE T}) : \text{ITYPE T} :=}
\]
\[
\text{match l with} \\
\text{| nil \Rightarrow V} \\
\text{| b::bs \Rightarrow pow bs (opt b V)} \\
\text{end.}
\]

Similarly to the functoriality of \( \text{opt} \) we mentioned earlier the multiple addition of variables is functorial. On morphisms the \( \text{pow} \) operation is defined by recursively applying the functoriality of \( \text{opt} \), where for the latter we employ a special notation with a prefixed hat.

\[
\text{Fixpoint pow_map (l : \mathbb{T}) V W (f : V \rightarrow W) :}
\]
\[
\text{V ** l \rightarrow W ** l :=}
\]
\[
\text{match l return V ** l \rightarrow W ** l with} \\
\text{| nil \Rightarrow f} \\
\text{| b::bs \Rightarrow pow_map (^f)} \\
\text{end.}
\]

In the same manner the multiple shifting

\[
\text{Fixpoint lshift (l : \mathbb{T}) (V W : \text{ITYPE T})}
\]
\[
(f : V \rightarrow P W) :}
\]
\[
\text{V ** l \rightarrow P (W ** l) := ...}
\]

is defined.

Representation  The implementation of representations and their morphisms now deviates from their construction on paper, where categorical facilities like the product on the category of modules and the fibre and derivative functors are heavily used. The reason becomes clear when we expand the commutative diagram for morphisms of representations. The four lower modules on the left side of the diagram all have the same carrier, but they are not equal as modules in Coq. We would hence have to insert isomorphisms as indicated in the diagram 3.1. This would result in quite a cumbersome formalization with decreased readability. Instead we use the possibilities of the type theory which underlies Coq. Given an arity \((s_i, t_i)_l \rightarrow t_0\) and a monad \(P\), we construct the module \( \prod_{i=1}^n (\partial s_i P)_{t_i} \) from scratch. Its carrier is given as an inductive type parametrized by a set of variables and an arity (resp. the relevant part of the arity), and the module substitution \text{mkleisli} is defined by recursion afterward.
\[
\prod_{i=1}^{n} \partial_{x_i} P_{t_i} \xrightarrow{p_{\alpha}} P_{t_0}
\]
modhom_from_arity P ((sig i),t).

Definition Repr := forall t, Repr_t t.

We bundle the data and define a representation as a monad together with a representation structure over this monad.

Record Representation := { rep_monad :> Monad (ITYPE T); repr : Repr rep_monad }.

Here an example of coercion occurs. The special notation :> allows to omit the projection rep_monad when accessing the monad of a given representation R. We can hence also write R x for the value of the monad of R on an object x of the underlying category. This is precisely the kind of abus de notation which is used in mathematics, and such coercions are used very often in our theory files.

Morphisms of representations Morphisms of representations are monad morphisms which satisfy a commutativity condition for each arity. As already mentioned, the carrier of the upper left product module is defined as an inductive type. This suggests the use of structural recursion for defining the left vertical morphism of the commutative diagram.

Fixpoint Prod_mor_c (l : \[\[T\] * T\]) (V : ITYPE T) (X : prod_mod P l V) :
f* (prod_mod Q l) V := match X in prod_mod_c _ _ l return f* (prod_mod Q l) V with | TTT => TTT _ _ | CONSTR b bs elem elems => CONSTR (f _ _ elem) (Prod_mor_c elems) end.

This function is easily proved to be a morphism of \(P\)-modules

\[
\text{Prod}_\text{mor} : \prod_{i=1}^n (\partial_{\vec{s}} P)_t \rightarrow f^* \prod_{i=1}^n (\partial_{\vec{s}} Q)_t.
\]

The isomorphism in the lower right corner however remains in the formalization, being called ITPB_FIB. For an arity a and module morphisms \(\text{RepP}\) and \(\text{RepQ}\) representing this arity over monads P and Q respectively, the definition of the commutative diagram reads as follows.

Definition commute f RepP RepQ : Prop := RepP ;; f [(snd a)] == Prod_mor (fst a) ;; f* RepQ ;; ITPB_FIB f _ _

A morphism of representations P and Q of the signature \(S\) is just a monad morphism from P to Q together with the commutativity property for each \(t : T\) and each arity (index) \(i\) in the indexing set \(S\ t\).
Variables P Q : Representation S.

Class Representation_Hom_s (f : Monad_Hom P Q) :=
  repr_hom_s : forall t (i : sig_index (S t)),
  commute f (repr P i) (repr Q i).

Record Representation_Hom : Type := {
  repr_hom_c :> Monad_Hom P Q;
  repr_hom :> Representation_Hom_s repr_hom_c
}.

Representations and their morphisms of a signature S form a category REPRESENTATION S. The formalization thereof is straightforward.

3.4 The initial object

The carrier STS of initial object of REPRESENTATION S is defined inductively together with a kind of product called STS_list, indexed by a set of variables and a list of argument types.

Inductive STS (V : I_TYPE T) : I_TYPE T :=
| Var : forall t, V t -> STS V t
| Build : forall t (i : sig_index (S t)),
  STS_list V (sig i) -> STS V t
with
STS_list (V : I_TYPE T) :
  [[T] * T] -> Type :=
| TT : STS_list V nil
| constr : forall b bs,
  STS (V ** (fst b)) (snd b) ->
  STS_list V bs ->
  STS_list V (b::bs).

The latter type is actually isomorphic to the type prod_mod_c STS. The use of mutual induction could hence have been avoided by defining a nested inductive type as follows.

Inductive STS (V : I_TYPE T) : I_TYPE T :=
| Var : forall t, V t -> STS V t
| Build : forall t (i : sig_index (S t)),
  prod_mod_c STS V (sig i) -> STS V t.

However, we preferred the mutual inductive version because it allows us to use two important Coq facilities. Firstly, in Coq mutually recursive functions can very easily be defined on mutually inductive data types. Secondly, a nice mutual induction scheme for mutual inductive types can automatically be generated by Coq using the Scheme command. Both of these are often used: all of our interesting functions – renaming, substitution and the initial morphisms – will be defined by mutual induction, and many properties of substitution and renaming, the so-called fusion properties will be proved using the special induction scheme. We hence decide that the benefits outweigh the small duplication of data. The
functions between the isomorphic data \texttt{STS\_list} and \texttt{prod\_mod\_c} \texttt{STS} are called \texttt{STS1\_f\_pm} and \texttt{pm\_f\_STS1}.

We then proceed by defining functions \texttt{rename\_shift} and \texttt{subst} providing functoriality, shifting and a substitution operation resp., and prove various fusion laws. Finally we establish that the function \texttt{subst} and the variable–as–term constructor \texttt{Var} turn \texttt{STS} into a monad.

Program Instance \texttt{STS\_monad} :
\begin{verbatim}
    Monad (ITYPE T) STS := {
        weta := Var;
        kleisli := subst
    }
\end{verbatim}

It remains to define the representational structure on \texttt{STS}. This structure is defined using the \texttt{Build} constructor. For each arity \texttt{i} in the index set \texttt{sig\_index (Sig t)} we must give a morphism of modules from \texttt{prod\_mod \texttt{STS} (sig i)} to \texttt{STS \texttt{[(t)]}}. Since the constructor \texttt{Build} takes its argument from \texttt{STS\_list} and not from \texttt{prod\_mod \texttt{STS}}, we precompose with one of the isomorphisms between those two types.

Program Instance \texttt{STS\_arity\_rep} (t : T) (i : sig\_index (S t)) :
\begin{verbatim}
    Module\_Hom
        (S := prod\_mod \texttt{STS (sig i)})
        (T := STS \texttt{[(t)]})
        (fun V X => Build (STS1\_f\_pm X)).
\end{verbatim}

The result is the object \texttt{STSRepr} of the category \texttt{REPRESENTATION S}. For a representation \texttt{R} of the signature \texttt{S} the initial morphism \texttt{init} from \texttt{STSRepr} to \texttt{R} is defined by mutual recursion.

Fixpoint \texttt{init} V t (v : STS V t) : R V t :=
\begin{verbatim}
match v in STS _ t return R V t with
    | Var t v => weta (Monad\_struct := R) V t v
    | Build t i X => repr R i V (init\_list X)
end
\end{verbatim}

where the function \texttt{init\_list} applies \texttt{init} to lists of arguments. Several lemmas then show that \texttt{init} commutes with renaming/lifting (\texttt{init\_lift}), shifting (\texttt{init\_shift}) and substitution (\texttt{init\_kleisli}). The latter property is precisely one of the axioms of morphisms of monads (see definition \texttt{2} rectangular diagram). The second monad morphism axiom which states compatibility with the \eta\_s of the monads involved is fulfilled by definition of \texttt{init} – it is exactly the first branch of the pattern matching. We hence have established that \texttt{init} is (the carrier of) a morphism of monads \texttt{init\_mon}. Very much less work is then needed to show that \texttt{init} also is a morphism of representations (cf. lemma \texttt{init\_rep}). Its uniqueness is expressed by the following lemma.

Lemma \texttt{init\_unique} :
\begin{verbatim}
    forall f : STSRepr ---> R , f == init\_rep.
\end{verbatim}
Proof.
   ...
Qed.

which is again proved with the help of the mutual induction scheme mentioned earlier. Finally we declare an instance of the Initial type class for the category of representations REPRESENTATION $S$ with STSRepr as initial object and init_rep $R$ as the initial morphism towards any other representation $R$.

Program Instance STS_initial :
   Initial (REPRESENTATION $S$) := {
      Init := STSRepr;
      InitMor $R$ := init_rep $R$
   }.

4 Conclusions & future work

We have presented a theory of representations of typed binding signatures in monads over (families of) sets. It features the relatively new notion of module over a monad and exhibits the structure of constructors as morphisms of modules.

Our theory does not use the traditional devices for syntax such as signature functors and their algebras. However, it is closely connected to the classical algebraic theory via an adjunction detailed by the second author [31].

We proved correct our theory in the proof assistant Coq. The theory naturally gives rise to a state–of–the–art implementation of syntax using intrinsic typing as advertised by [3]. Additionally it employs a new technique also coming directly from the theory, intrinsic environments. This means that variables are not elements of some untyped set, e. g. of the natural numbers. Instead they are elements of indexed sets, as is the case for intrinsically typed terms. As such the meta–language type of a variable carries the information about its object type.

The implementation may serve as a framework for language research similar to [6] with the difference of being based on Nested Abstract Syntax. Given a signature it generates the associated syntax, induction and recursion principles and substitution. The latter is certified to have the standard (monadic) fusion properties.

As already mentioned among related work, the first author is working on the extension of the theory of representations to account for compilation and reduction.

Another line of work is the extension of the theory to dependently typed syntax, which has been pursued by the second author [31].

References


[10] Fiore, M., Semantic analysis of normalisation by evaluation for typed lambda calculus, PPDP [DBL02], 26–37


[12] Fiore, M., Second-Order and Dependent-Sorted Abstract Syntax


[22] Huet, G. and Saïbi, A., Constructive Category Theory, Joint CLICS-TYPES Workshop on Categories and Type Theory, Göteborg, 1998


