Math. Struct. in Comp. Science (2016), *vol.* 26, *pp.* 3–37. © Cambridge University Press 2014
doi:10.1017/S0960129514000103 First published online 5 December 2014

Modules over relative monads for syntax and semantics

BENEDIKT AHRENS

Laboratoire J. A. Dieudonné, Université Nice Sophia Antipolis, France

Received January 2011; revised March 2012

We give an algebraic characterization of the syntax and semantics of a class of untyped functional programming languages.

To this end, we introduce a notion of *2-signature*: such a signature specifies not only the *terms* of a language, but also *reduction rules* on those terms. To any 2-signature (S, A) we associate a category of 'models'. We then prove that this category has an initial object, which integrates the terms freely generated by S, and which is equipped with reductions according to the rules given in A. We call this initial object the *programming language generated by* (S, A). Models of a 2-signature are built from *relative monads* and *modules* over such monads. Through the use of monads, the models – and in particular, the initial model – come equipped with a *substitution operation* that is compatible with reduction in a suitable sense.

The initiality theorem is formalized in the proof assistant Coq, yielding a machinery which, when fed with a 2-signature, provides the associated programming language with reduction relation and certified substitution.

1. Introduction

We give an algebraic characterization, via a universal property, of the programming language generated by a *signature*.

More precisely, we define a notion of *2-signature* which allows the specification of the *syntax* of a programming language – via a 1-signature, say, S – as well as its *semantics* in form of reduction rules, specified through a set A of "*inequations*" *over* S. To any 1-signature S we associate a category of *models of* S. Given a 2-signature (S, A), the inequations of A give rise to a *satisfaction* predicate on the models of S, and thus specify a full subcategory of models of S which satisfy the inequations of A. We call this subcategory the *category of models of* (S, A). Our main theorem states that this category has an initial object – the programming language associated to (S, A) – which integrates the terms generated by S, equipped with the reduction relation generated by the inequations of A.

The theorem has been fully checked in the proof assistant Coq (Coq 2010). The Coq theory files as well as documentation are available online at

`http://math.unice.fr/laboratoire/logiciels`.

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

1.1. *Summary*

We define a notion of 2-signature in order to specify the terms and reduction rules of functional programming languages. Given any 2-signature, we characterize its associated programming language as initial object in some category, thus giving an algebraic definition of abstract syntax with operational semantics. This characterization of syntax with reduction rules is given in two steps:

1. At first *pure syntax* is characterized as initial object in some category. Here, we use the term 'pure' to express the fact that no semantic aspects such as reductions on terms are considered. This characterization is actually a consequence of a result by Hirschowitz and Maggesi (2007a) (cf. Section 1.1.1).
2. Afterwards we consider *inequations* specifying *reduction rules*. Given a set of reduction rules for terms, we build up on the preceding result to give an algebraic characterization of syntax *with operational semantics*.

In summary, the merit of this work is to give an algebraic characterization of syntax *with reduction rules*, building up on such a characterization for *pure* syntax given by Hirschowitz and Maggesi (2007a).

Our approach is based on relative monads (Altenkirch *et al.* 2010) from the category Set of sets to the category Pre of preorders and modules over such monads. Compared to traditional monads, relative monads allow for different categories as domain and codomain.

We now explain the above two points in more detail:

1.1.1. *Pure syntax.* An *arity* specifies the type of a term constructor: it is given by a list of natural numbers, the length of which specifies the number of arguments of the constructor. The list entries specify the number of variables that are bound in the corresponding argument. A 1-signature is a family of arities. To any 1-signature S we associate a category $\mathsf{Rep}^{\Delta}(S)$ of *representations* – or 'models' – of S, where a model of S is built from a relative monad on the functor $\Delta : \mathsf{Set} \to \mathsf{Pre}$.

This category has an initial object (cf. Lemma 43), which integrates the terms freely generated by the signature. We call this object the *(pure) syntax associated to* S. As mentioned above, we use the term 'pure' to distinguish this initial object from the initial object associated to a *2-signature*, which gives an analogous characterization of syntax *with reduction rules* (cf. below).

Initiality for pure syntax is actually a consequence of a related initiality theorem proved by Hirschowitz and Maggesi (2007a): in that work, the authors associate, to any signature S, a category $\mathsf{Rep}(S)$ of models of S, where a model is built from a monad over Set. We connect the corresponding categories by exhibiting a pair of adjoint functors (cf. Lemma 42) between our category $\mathsf{Rep}^{\Delta}(S)$ of representations of S and that of Hirschowitz and Maggesi,

$$\mathsf{Rep}(S) \underset{U_*}{\overset{\Delta_*}{\rightleftarrows}} \mathsf{Rep}^{\Delta}(S) \quad (\perp).$$

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

We thus obtain an initial object in our category $\mathsf{Rep}^{\Delta}(S)$ using the fact that left adjoints are co-continuous: the image under the functor $\Delta_* : \mathsf{Rep}(S) \to \mathsf{Rep}^{\Delta}(S)$ of the initial object in the category $\mathsf{Rep}(S)$ is initial in $\mathsf{Rep}^{\Delta}(S)$.

1.1.2. *Syntax with reduction rules.* Given a 1-signature S, an *S-inequation* $E = (\alpha, \gamma)$ associates a pair (α^R, γ^R) of *parallel* morphisms in a suitable category to any representation R of S. In a sense made precise later, we can ask whether

$$\alpha^R \leqslant \gamma^R,$$

due to our use of relative monads into the category Pre of preorders. If this is the case, we say that R *satisfies* the inequation E. A *2-signature* is a pair (S, A) consisting of a 1-signature S, which specifies the terms of a language, together with a set A of *S-inequations*, which specifies reduction rules on those terms. Given a 2-signature (S, A), we call *representation of* (S, A) any representation of S that satisfies each inequation of A. The *category of representations of* (S, A) is defined to be the full subcategory of representations of S whose objects are representations of (S, A).

We would like to exhibit an initial object in the category of representations of (S, A), and thus must rule out inequations which are never satisfied. We call *classic* S-inequation any S-inequation whose codomain is of a particular form. Our main result states that for any set A of classic S-inequations the category of representations of (S, A) has an initial object. The class of classic inequations is large enough to account for the fundamental reduction rules; in particular, beta and eta reductions are given by classic inequations.

Our definitions ensure that any reduction rule between terms that is expressed by an inequation $E \in A$ is automatically propagated into subterms. The set A of inequations hence only needs to contain some 'generating' inequations, a fact that is well illustrated by our example 2-signature $\Lambda\beta$ of the untyped lambda calculus with beta reduction. This signature has only one inequation β which expresses beta reduction at the root of a term,

$$\lambda x.M(N) \quad \rightsquigarrow \quad M[x := N].$$

The initial representation of $\Lambda\beta$ is given by the untyped lambda calculus, equipped with the reflexive and transitive beta reduction relation $\twoheadrightarrow_\beta$ as presented by Barendregt and Barendsen (1994).

1.2. *Related work*

Initial semantics results for syntax with variable binding were first presented on the LICS'99 conference, see below. Those results are concerned only with the *syntactic aspect* of languages: they characterize the *set of terms* of a language as an initial object in some category, while not taking into account reductions on terms. In lack of a better name, we refer to this kind of initiality results as *purely syntactic*.

Some of these initiality theorems have been extended to also incorporate semantic aspects, e.g. in form of equivalence relations between terms. These extensions are reviewed in the second paragraph.

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

Purely syntactic results. Initial semantics for 'pure' syntax – i.e. without considering semantic aspects – with variable binding were presented by several people independently, differing in the modelling of variable binding.

The *nominal approach* by Gabbay and Pitts (1999) (see also Gabbay and Pitts (2001); Pitts (2003)) uses a set theory enriched with *atoms* to establish an initiality result. Their approach models lambda abstraction as a constructor which takes a pair of a variable name and a term as arguments. In contrast to the other techniques mentioned in this list, in the nominal approach syntactic equality is different from α-equivalence. Hofmann (1999) proves an initiality result modelling variable binding in a higher-order abstract syntax (HOAS) style. Fiore *et al.* (1999) (also Fiore (2002, 2005)) model variable binding through nested datatypes as introduced by Bird and Meertens (1998). Fiore *et al.*'s approach is extended to simply-typed syntax by Miculan and Scagnetto (2003). Tanaka and Power (2005) generalize and subsume those three approaches to a general category of contexts. An overview of this work and references to more technical papers is given by Power (2007). Hirschowitz and Maggesi (2007a) prove an initiality result for untyped syntax based on the notion of *module over a monad*. Their work has been extended to simply-typed syntax by Zsidó (2010).

Incorporating semantics. Rewriting in nominal settings has been examined by Fernández and Gabbay (2007). Ghani and Lüth (2003) present rewriting for algebraic theories without variable binding; they characterize equational theories (with a *symmetry* rule) resp. rewrite systems (with *reflexivity* and *transitivity* rule, but without *symmetry*) as *coequalizers* resp. *coinserters* in a category of monads on the categories Set resp. Pre. Fiore and Hur (2007) have extended Fiore's work to integrate semantic aspects into initiality results. In particular, Hur's thesis (2010) is dedicated to *equational* systems for syntax with variable binding. In a 'Further research' section (Hur 2010, Chap. 9.3), Hur suggests the use of preorders, or more generally, arbitrary relations to model *in*equational systems. Hirschowitz and Maggesi (2007a) prove initiality of the set of lambda terms modulo beta and eta conversion in a category of *exponential monads*. In an unpublished paper, Hirschowitz and Maggesi (2007b) define a notion of *half-equation* and *equation* to express congruence between terms. We adopt their definition in this paper, but interpret a pair of half-equations as *in*equation rather than equation. This emphasizes the *dynamic* viewpoint of reductions as *directed* equalities rather than the *static*, mathematical viewpoint one obtains by considering symmetric relations. In a 'future work' section, Hirschowitz and Maggesi (2010, Section 8) mention the idea of using preorders as an approach to model semantics, and they suggest interpreting the untyped lambda calculus with beta and eta reduction rule as a monad over the category Pre of preordered sets. The present work gives an alternative viewpoint to their suggestion by considering the lambda calculus with beta reduction – and a class of programming languages in general – as a preorder-valued *relative* monad on the functor $\Delta : \mathsf{Set} \to \mathsf{Pre}$. The rationale underlying our use of relative monads from sets to preorders is that we consider *contexts* to be given by unstructured sets, whereas *terms* of a language carry structure in form of a reduction relation. In this view it is reasonable to suppose variables and terms to live in *different* categories, which is possible through the use of relative monads on the functor $\Delta : \mathsf{Set} \to \mathsf{Pre}$ (cf.

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

Definition 3) instead of traditional monads (cf. also Remark 63). Relative monads were introduced by Altenkirch *et al.* (2010). In that work, the authors characterize the untyped lambda calculus as a relative monad over the inclusion functor from finite sets to sets. Their point of view can be combined with ours, leading to considering monads on the functor $\Delta \circ i : \mathsf{Fin} \to \mathsf{Pre}$, cf. Remark 62. Hirschowitz (2013), taking the viewpoint of categorical semantics, defines a category Sig of 2-signatures for simply-typed syntax with reduction rules, and constructs an adjunction between Sig and the category 2CCCat of small cartesian closed 2-categories. He thus associates to any signature a 2-category of types, terms and reductions satisfying a universal property. More precisely, terms are given by morphisms in this category, and reductions are expressed by the existence of 2-cells between terms. His approach differs from ours in the way in which variable binding is modelled: Hirschowitz encodes binding in a HOAS style through exponentials.

1.3. *Synopsis*

In the second section, we review the definition of (relative) monads and define modules over those monads as well as their morphisms. Some constructions on monads and modules are given, which will be of importance in what follows.

In the third section, we define arities, half-equations and inequations, as well as their representations. Afterwards we state our main result. The running example in these two sections is the 2-signature $\Lambda\beta$ of the lambda calculus with beta reduction.

In the fourth section, we describe some elements of the formalization of the main theorem in the proof assistant Coq.

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

2. Relative monads & modules

This section presents the category-theoretic structures from which our models of a signature are built.

At first we review relative monads as defined by Altenkirch *et al.* (2010) and define modules for those monads (cf. Section 2.1). Afterwards (cf. Sections 2.2 and 2.3), we port some constructions from modules over (traditional) monads, as presented, for instance, by Hirschowitz and Maggesi (2007a), Zsidó (2010) and Ahrens and Zsidó (2011), to modules over relative monads.

2.1. *Modules over relative monads*

We review the definition of relative monad as given by Altenkirch *et al.* (2010) and define suitable morphisms for them. As an example, we consider the lambda calculus as a relative monad from sets to preorders, on the functor $\Delta : \mathsf{Set} \to \mathsf{Pre}$. Finally, we define *modules* over relative monads and morphisms of such modules.

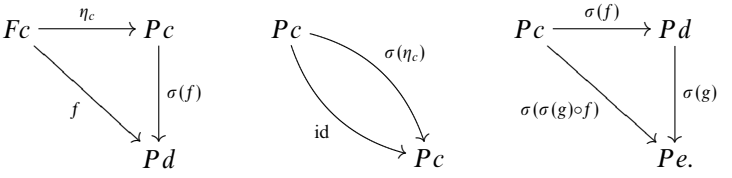
Definition 1 (relative monad). Given categories $\mathcal{C}$ and $\mathcal{D}$ and a functor $F : \mathcal{C} \to \mathcal{D}$, a *(relative) monad* $P : \mathcal{C} \xrightarrow{F} \mathcal{D}$ *on* F is given by

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

— a map $P : \mathcal{C}_0 \to \mathcal{D}_0$ on the objects of $\mathcal{C}$,
— for each object $c \in \mathcal{C}_0$, a morphism $\eta_c \in \mathcal{D}(Fc, Pc)$ and
— for all objects $c, d \in \mathcal{C}_0$ a *Kleisli operation*

$$\sigma_{c,d} : \mathcal{D}(Fc, Pd) \to \mathcal{D}(Pc, Pd)$$

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



Here and later we omit the object indices of the Kleisli operation.

Remark 2. A monad P is equipped with a functorial structure (also denoted by P) by setting, for a morphism $f : a \to b$ in $\mathcal{C}$,

$$P(f) := \mathrm{lift}_P(f) := \sigma(\eta \circ Ff),$$

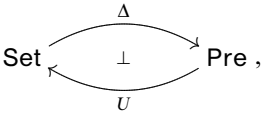
the functoriality axioms being a consequence of the monad axioms.

We are mainly interested in monads on a specific functor:

Definition 3 ($\Delta : \mathsf{Set} \to \mathsf{Pre}$). We call $\Delta : \mathsf{Set} \to \mathsf{Pre}$ the functor from sets to preordered sets which associates to each set X the set itself together with the smallest preorder, i.e. the diagonal of X,

$$\Delta(X) := (X, \delta_X).$$

The functor Δ is a full *embedding*, i.e. it is fully faithful and injective on objects. Furthermore it is left adjoint to the forgetful functor $U : \mathsf{Pre} \to \mathsf{Set}$,



that is, the embedding $\Delta : \mathsf{Set} \to \mathsf{Pre}$ is a coreflection. We denote by φ the family of isomorphisms

$$\varphi_{X,Y} : \mathsf{Pre}(\Delta X, Y) \cong \mathsf{Set}(X, UY).$$

We omit the indices of φ whenever they can be deduced from the context.

Lemma 4 (monads over Δ and monads on Set). Let P be a monad on Δ. By postcomposing with the forgetful functor $U : \mathsf{Pre} \to \mathsf{Set}$ we obtain a monad

$$\bar{P} : \mathsf{Set} \to \mathsf{Set}.$$

The substitution is defined, for $f \in \mathsf{Set}(a, UPb)$ by setting

$$\sigma^{\bar{P}}(f) := U\left(\sigma(\varphi^{-1} f)\right),$$

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

making use of the adjunction $f \mapsto \varphi^{-1} f \in \mathsf{Pre}(\Delta a, P b)$ of Definition 3. Conversely, to any monad Q on Set we associate a relative monad by postcomposing with the functor Δ:

$$\Delta Q : \mathsf{Set} \xrightarrow{\Delta} \mathsf{Pre}.$$

The above construction in Lemma 4 actually is an instance of a general lemma: given an adjunction $F \dashv G$ such that $GF = \mathrm{Id}_C$, one can associate a relative monad on F to any monad on $\mathcal{C}$ and vice versa. These maps are functorial, and one obtains an adjunction between a suitable category of relative monads on F and a category of monads on $\mathcal{C}$. Details will be reported elsewhere.

Example 5. Consider the set of all lambda terms indexed by their set of free variables

$$\begin{aligned} \mathrm{LC}(V) ::= \quad & \mathrm{Var} : V \to \mathrm{LC}(V) \\ \mid \ & \mathrm{Abs} : \mathrm{LC}(V') \to \mathrm{LC}(V) \\ \mid \ & \mathrm{App} : \mathrm{LC}(V) \times \mathrm{LC}(V) \to \mathrm{LC}(V), \end{aligned}$$

where $V' := V + \{*\}$ is the set V enriched with a new distinguished element, i.e. a context extended by one additional free variable. We occasionally write λ for Abs and denote application by juxtaposition. Altenkirch and Reus (1999) interpret LC as a monad over sets. Altenkirch *et al.* (2010) interpret LC as a relative monad on the inclusion functor $i : \mathsf{Fin} \to \mathsf{Set}$, by restricting contexts to be given by *finite* sets (cf. also Remark 62).

We equip each $\mathrm{LC}(V)$ with a preorder taken as the reflexive–transitive closure of the relation generated by the rule

$$\lambda M(N) \leqslant M[* := N] \qquad (\beta)$$

and its propagation into subterms. This defines a monad from sets to preorders

$$\mathrm{LC}\beta : \mathsf{Set} \xrightarrow{\Delta} \mathsf{Pre}.$$

The family η^{LC} is given by the constructor Var, and the Kleisli operation

$$\sigma_{X,Y} : \mathsf{Pre}\big(\Delta X, \mathrm{LC}\beta(Y)\big) \to \mathsf{Pre}\big(\mathrm{LC}\beta(X), \mathrm{LC}\beta(Y)\big)$$

is given by simultaneous substitution. Via the adjunction of Definition 3 the substitution can also be read as

$$\sigma_{X,Y} : \mathsf{Set}\big(X, \mathrm{LC}(Y)\big) \to \mathsf{Pre}\big(\mathrm{LC}\beta(X), \mathrm{LC}\beta(Y)\big).$$

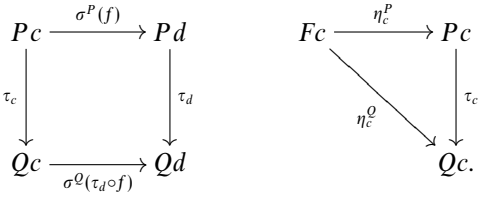
The substitution can hence be chosen as for the monad LC, but one has to prove the additional property of monotonicity in the first-order argument.

Inspired by the example above we sometimes call the Kleisli operation of a monad *monadic substitution*, even for monads that do not denote the terms of a language over free variables.

For two monads P and Q from $\mathcal{C}$ to $\mathcal{D}$ a *morphism of monads* is a family of morphisms $\tau_c \in \mathcal{D}(Pc, Qc)$ that is compatible with the monadic structure:

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

Definition 6 (morphism of relative monads). Given two relative monads P and Q from $\mathcal{C}$ to $\mathcal{D}$ on the same functor $F : \mathcal{C} \to \mathcal{D}$, a *morphism of monads* from P to Q is given by a collection of morphisms $\tau_c \in \mathcal{D}(Pc, Qc)$ such that the following diagram commutes for any suitable morphism f:

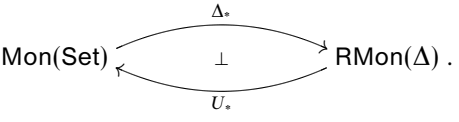


As a consequence from these commutativity properties the family τ is a natural transformation between the functors induced by the monads P and Q.

Monads on $F : \mathcal{C} \to \mathcal{D}$ and their morphisms form a category $\mathsf{RMon}(F)$ where identity and composition of morphisms are defined by pointwise identity and composition of morphisms. We have a similar category of traditional monads:

Definition 7 (category of monads). We denote by $\mathsf{Mon}(\mathcal{C})$ the category of monads on $\mathcal{C}$ and morphisms of such monads. More precisely, a morphism $f : P \to Q$ in $\mathsf{Mon}(\mathcal{C})$ is given by a family $\tau_c : \mathcal{C}(Pc, Qc)$ of morphisms that is compatible with the monadic structure, analogously to the diagrams of Definition 6.

Lemma 8 (adjunction between $\mathsf{RMon}(\Delta)$ and $\mathsf{Mon}(\mathsf{Set})$). The maps defined in Lemma 4 give rise to an adjunction between the category of monads over Δ and the category of monads over sets, where the functor U_* is defined on objects as in Lemma 4 by $U_*(R) := \bar{R}$:



Informally, the notion of relative monad on a base functor F is obtained from the notion of monad (in Manes style, i.e. with Kleisli operation) by inserting applications of the functor F where necessary (Altenkirch *et al.* 2010). Similarly, one obtains the notion of module over a relative monad from the notion of module over a monad – in form of a Kleisli operation as presented, for instance, by Ahrens and Zsidó (2011) – by inserting applications of the base functor F:

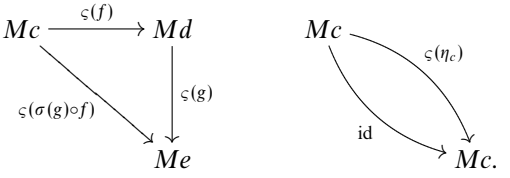
Definition 9 (module over relative monad). Let $P : \mathcal{C} \xrightarrow{F} \mathcal{D}$ be a relative monad and let $\mathcal{E}$ be a category. A *relative module M over P with codomain $\mathcal{E}$* is given by

— a map $M : \mathcal{C}_0 \to \mathcal{E}_0$ on the objects of the categories involved and
— for all objects $c, d \in \mathcal{C}_0$ a map

$$\varsigma_{c,d} : \mathcal{D}(Fc, Pd) \to \mathcal{E}(Mc, Md)$$

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

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



Functoriality for such a module M is defined in a similar way as for monads, by setting, for any morphism $f : c \to d$ in $\mathcal{C}$,

$$M(f) := \mathrm{rmlift}_M(f) := \varsigma(\eta \circ Ff).$$

Example 10 (monoids and monoid actions). An anonymous referee suggested the following example: let $F : 1 \to \mathsf{Set}$ be the functor on the one-object category which maps to the final object, i.e. the singleton set. Then a relative monad on F is a monoid. Given a relative monad G on F, a G-module with codomain Set is a monoid action of G.

The following examples of modules are instances of more general constructions explained in the next section:

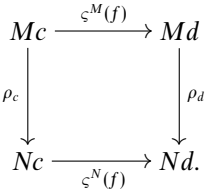
Example 11 (tautological module). The map $\mathrm{LC}\beta : V \mapsto \mathrm{LC}\beta(V)$ yields a module over the monad $\mathrm{LC}\beta$, the *tautological module* $\mathrm{LC}\beta$.

Example 12 (derived module). Let $V' := V + \{*\}$. The map $\mathrm{LC}\beta' : V \mapsto \mathrm{LC}\beta(V')$ inherits the structure of an $\mathrm{LC}\beta$-module from the tautological $\mathrm{LC}\beta$-module $\mathrm{LC}\beta$ of Example 11. We call $\mathrm{LC}\beta'$ the *derived module* of the module $\mathrm{LC}\beta$.

Example 13 (product of modules). The map $V \mapsto \mathrm{LC}\beta(V) \times \mathrm{LC}\beta(V)$ inherits an $\mathrm{LC}\beta$-module structure from $\mathrm{LC}\beta$.

A *module morphism* is given by a family of morphisms that is compatible with module substitution:

Definition 14 (module morphism). Let P be a relative monad on $F : \mathcal{C} \to \mathcal{D}$, and let M and N be two relative modules over P with codomain $\mathcal{E}$. A *morphism of relative P-modules* from M to N is given by a collection of morphisms $\rho_c \in \mathcal{E}(Mc, Nc)$ such that for any morphism $f \in \mathcal{D}(Fc, Pd)$ the following diagram commutes:



The modules over P with codomain $\mathcal{E}$ and morphisms between them form a category called $\mathrm{RMod}(P, \mathcal{E})$. Note that the 'R' here stands for 'relative', not for 'right' as opposed to 'left'.

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

Composition and identity morphisms of modules are defined by pointwise composition and identity, similarly to the category of monads.

Example 15 (Examples 11, 12, 13 cont.). Abstraction and application are morphisms of LCβ-modules,

$$\mathrm{Abs} : \mathrm{LC}\beta' \to \mathrm{LC}\beta,$$
$$\mathrm{App} : \mathrm{LC}\beta \times \mathrm{LC}\beta \to \mathrm{LC}\beta.$$

2.2. *Constructions on monads and modules*

The constructions on modules over monads as used by Hirschowitz and Maggesi (2007a) carry over to relative monads:

Definition 16 (tautological module). Given a monad P on $F : \mathcal{C} \to \mathcal{D}$, we define the *tautological module* (also denoted by P) over P to be the module $(M, \varsigma) := (P, \sigma)$, i.e. with object map P and module substitution given by the monad substitution. Thus, the monad P can be considered as an object in the category $\mathrm{RMod}(P, \mathcal{D})$.

Definition 17 (constant and terminal module). Let P be a monad on $F : \mathcal{C} \to \mathcal{D}$. For any object $e \in \mathcal{E}$ the constant map $T_e : \mathcal{C} \to \mathcal{E}$, $c \mapsto e$ for all $c \in \mathcal{C}$ yields a P-module (T_e, id). In particular, if $\mathcal{E}$ has a terminal object $1_{\mathcal{E}}$, then the constant module $(T_{1_{\mathcal{E}}}, \mathrm{id})$ is terminal in $\mathrm{RMod}(P, \mathcal{E})$.

Definition 18 (postcomposition with a functor). Let P be a monad on $F : \mathcal{C} \to \mathcal{D}$, and let M be a P-module with codomain $\mathcal{E}$. Let $G : \mathcal{E} \to \mathcal{X}$ be a functor. Then the object map $G \circ M : \mathcal{C} \to \mathcal{X}$ defined by $c \mapsto G(M(c))$ is equipped with a P-module structure by setting, for $c, d \in \mathcal{C}$ and $f \in \mathcal{D}(Fc, Pd)$,

$$\varsigma^{G \circ M}(f) := G(\varsigma^{M}(f)).$$

For $M := P$ and G a constant functor mapping to an object $x \in \mathcal{X}$ and its identity morphism id_x, we obtain the constant module (T_x, id) as in Definition 17.

Let P and Q be two monads on $F : \mathcal{C} \to \mathcal{D}$. Given a monad morphism $h : P \to Q$, we can turn any Q-module M into a P-module, by 'pulling it back' along h:

Definition 19 (pullback module). Let $h : P \to Q$ be a morphism of monads on $F : \mathcal{C} \to \mathcal{D}$ and let M be a Q-module with codomain $\mathcal{E}$. We define a P-module h^*M to $\mathcal{E}$ with object map $c \mapsto Mc$ 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 extends to module morphisms and is functorial.

Remark 20. The pullback P-module h^*M has the same underlying functor as the Q-module M. It is merely the substitution action that changes: while ς^M expects morphisms in $\mathcal{C}(Fc, Qd)$ as arguments, the substitution of h^*M expects morphisms in $\mathcal{C}(Fc, Pd)$.

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

Definition 21 (induced module morphism). With the same notation as before, the monad morphism h induces a morphism of P-modules $h\colon P \to h^*Q$.

Remark 22. Note that the preceding two constructions do not change the functor resp. natural transformation underlying the module resp. morphism of modules.

The following construction explains Example 13:

Definition 23 (products). Suppose the category $\mathcal{E}$ is equipped with a product. Let M and N be P-modules with codomain $\mathcal{E}$. Then the map (on objects)

$$\mathcal{C} \to \mathcal{E}, \quad c \mapsto Mc \times Nc$$

is equipped with a module substitution by setting

$$\varsigma^{M\times N}(f) := \varsigma^M(f) \times \varsigma^N(f).$$

This construction extends to a product on $\mathrm{RMod}(P, \mathcal{E})$.

There is also a category where modules over different monads are grouped together:

Definition 24. Let $\mathcal{C}, \mathcal{D}$ and $\mathcal{E}$ be categories and $F : \mathcal{C} \to \mathcal{D}$ be a functor. We define the category $\mathrm{LMod}(F, \mathcal{E})$ ('L' for 'large') to be the category whose objects are pairs (R, M), where R is a monad over $F : \mathcal{C} \to \mathcal{D}$ and M is an R-module. A morphism from (R, M) to (S, N) is a pair (ρ, τ) where $\rho : R \to S$ is a monad morphism and τ is an R-module morphism $\tau : M \to \rho^* N$.

We are particularly interested in monads over the functor $\Delta : \mathsf{Set} \to \mathsf{Pre}$. The following construction – *derivation* – applies to modules over such monads.

2.3. *Derived modules*

Roughly speaking, a binding constructor makes free variables disappear. Its input are hence terms in an extended context, i.e. with (one or more) additional free variables compared to the output. *Derivation* is about context extension.

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

$$V' := V + \{*\}$$

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. Given a map $f : V \to W$ and $w \in W$, we call

$$\mathrm{default}(f, w) : V' \to W$$

the coproduct map defined by

$$\mathrm{default}(f, w) = [f, \lambda x.w].$$

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

Definition 25 (derived module). Given a monad P on $\Delta : \mathsf{Set} \to \mathsf{Pre}$ and a P-module M with codomain $\mathcal{E}$, we define the *derived module* by setting

$$M'(V) := M(V').$$

For a morphism $f \in \mathsf{Pre}(\Delta V, PW)$ the module substitution for the derived module is given by

$$\varsigma^{M'}(f) := \varsigma^{M}(\mathrm{shift}(f)).$$

Here the shifted map

$$\mathrm{shift}(f) \in \mathsf{Pre}(\Delta(V'), P(W'))$$

is defined via the adjunction of Definition 3 as

$$\mathrm{shift}(f) := \varphi^{-1}\Big(\mathrm{default}\big(P(\mathrm{inl}) \circ f, \eta(\mathrm{inr}(*))\big)\Big),$$

where $[\mathrm{inl}, \mathrm{inr}] = \mathrm{id}_{W'}$.

Derivation extends to an endofunctor on the category of P-modules with codomain $\mathcal{E}$.

Remark 26. When P is a monad of terms over free variables, the map $\mathrm{shift}(f)$ sends the additional variable of V' to $\eta^P(*)$, i.e. to the term consisting of just the 'freshest' free variable. When recursively substituting with a map $f : V \to P(W)$, terms under a binder such as λ must be substituted with the map $\mathrm{shift}(f)$.

Definition 27. Given a natural number n, we write M^n for the module M derived n times. Given a list $s = [n_1, \ldots, n_m]$ of natural numbers, we write $M^s := M^{n_1} \times \cdots \times M^{n_m}$.

Product and derivation are functorial, and we use the same notation *on morphisms*. That is, given a morphism of P-modules $\rho : M \to N$, we write

$$\rho^s := \rho^{n_1} \times \cdots \times \rho^{n_m} : M^s \to N^s.$$

The pullback operation commutes with products and derivations :

Lemma 28. Let $\mathcal{C}$ and $\mathcal{D}$ be categories and $\mathcal{E}$ be a category with products. Let P and Q be monads on $F : \mathcal{C} \to \mathcal{D}$ and $\rho : P \to Q$ a monad morphism. Let M and N be P-modules with codomain $\mathcal{E}$. The pullback functor is cartesian:

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

Lemma 29. Let P be a relative monad on $\Delta : \mathsf{Set} \to \mathsf{Pre}$ and M a module over P with codomain $\mathcal{E}$. Then we have

$$\rho^*(M') \cong (\rho^* M)'.$$

Definition 30 (substitution of *one* variable). We denote by wPre ('w' for 'weak') the category whose objects are preordered sets, and where a morphism from A to B is a map between the underlying sets, that is, a morphism $UA \to UB$ in Set.

Given a monad $P : \mathsf{Set} \to \mathsf{Pre}$ on Δ, and a P-module M with codomain Pre, we can consider M as a P-module with codomain wPre by postcomposing with the injection. We

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

denote this module by $\hat{M} \in \mathrm{RMod}(P, \mathsf{wPre})$. For any set X, we define the substitution of just one variable,

$$\begin{aligned} \mathrm{subst}_X : P(X') \times P(X) &\to P(X), \\ (y, z) &\mapsto y[* := z] := \sigma(\mathrm{default}(\eta_X, z))(y). \end{aligned}$$

This defines a morphism of P-modules with codomain wPre,

$$\mathrm{subst}^P : \hat{P}' \times \hat{P} \to \hat{P}.$$

Remark 31. Note that the substitution module morphism defined above is by construction monotone in its first argument, but not in its second argument. This is the reason why we cannot consider subst^P as a morphism of P-modules

$$\mathrm{subst}^P : P' \times P \to P,$$

but have to switch to the category wPre. This fact and a way to ensure monotonicity also in the second argument are explained more generally in Remark 61.

3. 2-signatures and their representations

In this section we define *2-signatures* and their representations. We then prove that, given any 2-signature, its associated category of representations has an initial object, the language generated by the 2-signature.

An *arity* describes the number of arguments and binding behaviour of a constructor of a syntax. A *1-signature* S is a family of arities and as such specifies the terms of a language. An *inequation over* S – also called *S-inequation* – expresses a reduction rule on the terms of the syntax associated to S. A *2-signature* is given by a 1-signature S and a set of S-inequations.

We define *representations* of a 1-signature S analogously to Hirschowitz and Maggesi (2007a) and Zsidó (2010), except that we use *relative* monads and modules over such monads. Afterwards we use *inequations over* S to specify reduction rules, and we consider those representations of S that satisfy the given inequations. We show that among those representations there is an initial representation, which integrates the terms generated by S, equipped with reductions according to the given inequations. Our inequations are precisely Hirschowitz and Maggesi's (2007b) *equations*, i.e. parallel pairs of half-equations. We simply interpret such a pair to define a *(directed) reduction* rather than an equality. Throughout this section the running example is the 2-signature $\Lambda\beta$ which specifies the lambda calculus with beta reduction.

3.1. *Arities, 1-signatures and their representations*

We first consider pure syntax, i.e. syntax without reductions. We specify syntax by a *1-signature*:

Definition 32 (arity, 1-signature). An *arity* is a list of natural numbers. A *1-signature* is a family of arities.

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

Intuitively, the length of an arity specifies the number of arguments of a constructor, and the ith entry of the arity specifies the number of variables which are bound by the constructor in the ith argument.

Example 33. The 1-signature Λ of untyped lambda calculus is given by the two arities

$$\mathrm{app} := [0,0], \quad \mathrm{abs} := [1].$$

Definition 34. Let $s := [n_1, n_2, \ldots, n_m]$ be an arity and P be a monad on the functor $\Delta : \mathsf{Set} \to \mathsf{Pre}$. We call

$$\mathrm{dom}(s, P) := P^s = P^{n_1} \times \cdots \times P^{n_m}$$

the *domain module* of s for P. Note that we use the notation defined in Definition 27.

Definition 35 (representations of a 1-signature). A *representation R of a 1-signature S* is given by a monad P over the functor $\Delta : \mathsf{Set} \to \mathsf{Pre}$ and, for each arity $s = [n_1, n_2, \ldots, n_m] \in S$, a morphism of P-modules

$$s^R : \mathrm{dom}(s, P) \to P.$$

Given a representation R, we denote its underlying monad by R as well.

Example 36 (Example 33 continued). A representation R of the 1-signature Λ is given by

— a monad $R : \mathsf{Set} \xrightarrow{\Delta} \mathsf{Pre}$ and
— two morphisms of R-modules in $\mathrm{RMod}(R, \mathsf{Pre})$,

$$\mathrm{app}^R : R \times R \to R \quad \text{and} \quad \mathrm{abs}^R : R' \to R.$$

Remark 37. A representation of a 1-signature à la Hirschowitz and Maggesi (2007a) is defined analogously, except for the use of (plain) monads on the category of sets and modules thereon instead of *relative* monads and modules on relative monads.

Definition 38. To any representation R of a 1-signature S in a relative monad R as defined in Definition 35 we associate a representation $U_*(R)$ of S in the monad $\bar{R}$ (cf. Lemma 4) in the sense of Hirschowitz and Maggesi (2007a) by postcomposing the representation module morphism of any arity s of S with the forgetful functor from preorders to sets:

$$s^R : R^s \to R \quad \mapsto \quad s^{U_* R} : \bar{R}^s \to \bar{R},$$

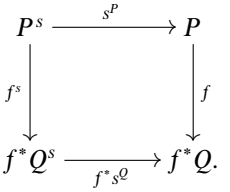
where we use $(\bar{R}^s) = (\bar{R})^s$. Conversely, to any representation Q of S in a monad Q over sets we can associate a representation $\Delta_* Q$ of S in the relative monad $\Delta_* Q$ on Δ, by postcomposing the representation module morphisms with Δ.

Morphisms of representations are monad morphisms which commute with the representation morphisms of modules:

Definition 39 (morphism of representations). Let P and Q be representations of a signature S. A *morphism of representations* $f : P \to Q$ is a morphism of monads $f : P \to Q$ such

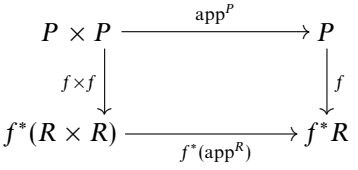
https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

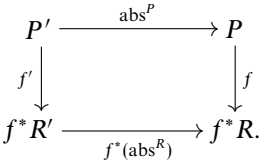
that the following diagram commutes for any arity s of S:



Note that we make extensive use of the notation defined in Definition 27. To make sense of this diagram it is necessary to recall the constructions on modules of Section 2.2. The diagram lives in the category $\mathrm{RMod}(P, \mathsf{Pre})$. The vertices are obtained from the tautological modules P resp. Q over the monads P resp. Q by applying the derivation and pullback functors as well as by the use of the product in the category of P-modules into Pre. The vertical morphisms are module morphisms induced by f, to which functoriality of derivation and products are applied. Furthermore, instances of Lemmas 28 and 29 are hidden in the lower left corner. The lower horizontal morphism makes use of the functoriality of the pullback operation.

Example 40 (Example 36 continued). Let P and R be two representations of Λ. A *morphism* from P to R is given by a morphism of monads $f : P \to R$ such that the following diagrams of P-module morphisms commute:



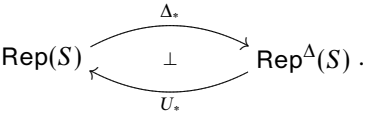


Composition and identity morphisms of representations are given by composition and identity of monad morphisms. We obtain a category of representations:

Definition 41 (category of representations). Representations of S and their morphisms form a category $\mathsf{Rep}^{\Delta}(S)$.

Since we are not considering any reductions on terms yet, but only plain syntax, it comes as no surprise that, for any 1-signature S, our category of representations of S relates to Hirschowitz and Maggesi's:

Lemma 42. The assignment of Definition 38 extends to an adjunction between our category of representations $\mathsf{Rep}^{\Delta}(S)$ in relative monads on Δ and Hirschowitz and Maggesi's category $\mathsf{Rep}(S)$ of representations in monads over sets:



https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

Lemma 43. Given a signature S, the category of representations of S in relative monads on Δ has an initial representation. Its underlying monad associates to any set V of variables the set of terms of the language specified by S in the context V, equipped with the diagonal preorder.

Proof. This is a direct consequence of the fact that left adjoints preserve colimits (Mac Lane 1998, Chapter V.5), thus, in particular, initial objects. □

Remark 44. The formalization in Coq of Lemma 43 (cf. Section 4) does not appeal to Hirschowitz and Maggesi's result, but constructs the initial object from scratch.

3.2. *Inequations over 1-signatures*

Consider the beta rule of lambda calculus,

$$\lambda M(N) \rightsquigarrow M[* := N].$$

We would like to express such a rule through a suitable inequation. In our formalism, abstraction and application are considered as morphisms of modules (cf. Example 15), and so is substitution (cf. Definition 30). This suggest to define (in)equations over a 1-signature S as *parallel pairs* of module morphisms, indexed by representations of S. Put differently, an (in)equation associates a parallel pair of module morphisms to any representation of S. Hirschowitz and Maggesi (2007b) specify equations through such pairs of (indexed) module morphisms over (plain) monads. We adapt their definition to our use of *relative* monads and modules over such monads. Afterwards we simply interpret a pair of half-equations as *inequation* rather than equation.

Definition 45 (category of half–equations). Let S be a signature. An *S-module* U is a functor from the category of representations of S to the category of modules $\mathrm{LMod}(\Delta, \mathsf{wPre})$ commuting with the forgetful functor to the category of relative monads on Δ:

$$\begin{array}{ccc} \mathsf{Rep}^{\Delta}(S) & \xrightarrow{\quad U \quad} & \mathrm{LMod}(\Delta, \mathsf{wPre}) \\ & \searrow \quad \swarrow & \\ & \mathsf{RMon}(\Delta). & \end{array}$$

We define a morphism of S-modules to be a natural transformation which becomes the identity when composed with the forgetful functor. We call these morphisms *half-equations*. The collection of S-modules and their morphisms yield a category which we call the category of S-modules (or the category of half-equations).

Remark 46. Objects of $\mathrm{LMod}(\Delta, \mathsf{wPre})$ are pairs of a monad P on $\Delta : \mathsf{Set} \to \mathsf{Pre}$ and a P-module M. Given an S-module U, we sometimes write $U(R)$ for the second component of $U(R)$, i.e. for the module over the monad (underlying the representation) R, see for instance Remark 47. We also write

$$U_X^R := U(R)(X)$$

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

for the value of an S-module at the representation R and the set X. Similarly, for a half-equation $\alpha : U \to V$ we write

$$\alpha_X^R := \alpha(R)(X) : U_X^R \to V_X^R \ .$$

Remark 47. A half-equation α from S-module U to V associates to any representation P a morphism of P-modules $\alpha^P : U(P) \to V(P)$ in $\mathrm{RMod}(P, \mathsf{wPre})$ such that for any morphism of S–representations $f : P \to R$ the following diagram commutes:

$$\begin{array}{ccc} (P, U(P)) & \xrightarrow{\alpha^P} & (P, V(P)) \\ \downarrow{\scriptstyle (f, U(f))} & & \downarrow{\scriptstyle (f, V(f))} \\ (R, f^*(U(R))) & \xrightarrow[\alpha^R]{} & (R, f^*(V(R))). \end{array}$$

Lemma 48. The category of S-modules is cartesian.

We give some examples of generic S-modules. The inductive class of S-modules thus defined is of importance later.

Definition 49 (classic S-module). We call *classic* any S-module satisfying the following inductive predicate:

— The map $\Theta : R \mapsto (R, \hat{R})$ is a classic S-module.
— If the S-module $M : R \mapsto (M_1(R), M_2(R))$ is classic, so is

$$M' : R \mapsto \big(M_1(R), M_2(R)'\big).$$

— If M and N are classic, so is

$$M \times N : R \mapsto \big(M_1(R), M_2(R) \times N_2(R)\big).$$

— The terminal module $* : R \mapsto (R, 1)$ is classic.

Using the same notation as in Definition 27, any list of natural numbers defines uniquely a classic S-module.

The following examples of half-equations are building blocks for the inequation specifying beta reduction:

Definition 50. The substitution operation

$$\mathrm{subst} : R \mapsto \mathrm{subst}^R : \hat{R}' \times \hat{R} \to \hat{R}$$

is a half-equation over any signature S. Its domain and codomain are classic.

Example 51 (Example 33 continued). The map

$$\mathrm{app} \circ (\mathrm{abs} \times \mathrm{id}) : R \mapsto \mathrm{app}^R \circ (\mathrm{abs}^R \times \mathrm{id}^R) : \hat{R}' \times \hat{R} \to \hat{R}$$

is a half-equation over the signature Λ.

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

Definition 52. Any arity $s = [n_1, \ldots, n_m] \in S$ defines a classic S-module

$$\mathrm{dom}(s) : R \mapsto R^s.$$

An *inequation* is a pair of parallel half-equations:

Definition 53 (inequation, 2–signature). Given a 1-signature S, an *S-inequation* is a pair of parallel half-equations between S-modules. We write

$$\alpha \leqslant \gamma : U \to V$$

for the inequation (α, γ) with domain U and codomain V. A *2-signature* is a pair (S, A) of a 1-signature S and a set A of S-inequations.

Given a set A of S-inequations, we can ask whether a given representation R of S satisfies the inequations of A. If this is the case, we call R a representation of the 2-signature (S, A):

Definition 54 (representations of a 2–signature). Let $\alpha \leqslant \gamma : U \to V$ be an inequation over S, and let R be a representation of S. We say that R *satisfies* $\alpha \leqslant \gamma$ if $\alpha^R \leqslant \gamma^R$ pointwise, i.e. if for any set X and any $y \in U(R)(X)$,

$$\alpha^R_X(y) \leqslant \gamma^R_X(y).$$

For a set A of S-inequations, we call *representation of* (S, A) any representation of S that satisfies each inequation of A. We define the category of representations of the 2-signature (S, A) to be the full subcategory in the category of representations of S whose objects are representations of (S, A).

Example 55 (Example 51 continued). We denote by β the Λ-inequation

$$\mathrm{app} \circ (\mathrm{abs} \times \mathrm{id}) \leqslant \mathrm{subst}\,. \tag{β}$$

We call $\Lambda\beta$ the 2-signature $((\mathrm{app}, \mathrm{abs}), \beta)$. A representation P of $\Lambda\beta$ is given by

— a monad $P : \mathsf{Set} \xrightarrow{\Delta} \mathsf{Pre}$ and
— two morphisms of P-modules

$$\mathrm{app} : P \times P \to P \quad \text{and} \quad \mathrm{abs} : P' \to P$$

such that for any set X and any $y \in P(X')$ and $z \in PX$

$$\mathrm{app}_X(\mathrm{abs}_X(y), z) \leqslant y[* := z].$$

3.3. *Initiality for 2–signatures*

Given a 2-signature (S, A), we would like to exhibit an initial object in its associated category of representations. However, we have to rule out inequations which are never satisfied. We restrict ourselves to inequations with a classic codomain:

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

Definition 56 (classic inequation). We say that an S-inequation is *classic* if its codomain is classic.

Theorem 57. For any set of classic S-inequations A, the category of representations of (S, A) has an initial object.

Proof. The basic ingredients for building the initial representation are given by the initial representation $\Delta_*\Sigma$ in the category $\mathsf{Rep}^{\Delta}(S)$ (cf. Lemma 43) or, equivalently, by the initial representation Σ in $\mathsf{Rep}(S)$. We call Σ the monad underlying the representation Σ.

The proof consists of 3 steps: at first, we define a preorder $\leqslant_A$ on the terms of Σ, induced by the set A of inequations. Afterwards we show that the data of the representation Σ – substitution, representation morphisms etc. – is compatible with the preorder $\leqslant_A$ in a suitable sense. This will yield a representation Σ_A of (S, A). Finally we show that Σ_A is the initial such representation.

— *The monad underlying the initial representation.*

For any set X, we equip ΣX with a preorder A by setting, for $x, y \in \Sigma X$,

$$x \leqslant_A y \quad :\Leftrightarrow \quad \forall R : \mathsf{Rep}(S, A), \quad i_R(x) \leqslant_R i_R(y), \tag{3.1}$$

where $i_R : \Sigma \to \bar{R}$ is the initial morphism of representations in monads coming from Zsidó's theorem (or, equivalently, the initial morphism $\Delta_*\Sigma \to R$). We have to show that the map

$$X \mapsto \Sigma_A X := (\Sigma X, \leqslant_A)$$

yields a relative monad over Δ. The missing fact to prove is that the substitution of the monad Σ with a morphism

$$f \in \mathsf{Pre}(\Delta X, \Sigma_A Y) \cong \mathsf{Set}(X, \Sigma Y)$$

is compatible with the order $\leqslant_A$: given any $f \in \mathsf{Pre}(\Delta X, \Sigma_A Y)$ we show that $\sigma^{\Sigma}(f) \in \mathsf{Set}(\Sigma X, \Sigma Y)$ is monotone with respect to $\leqslant_A$ and hence (the carrier of) a morphism $\sigma(f) \in \mathsf{Pre}(\Sigma_A X, \Sigma_A Y)$. We overload the infix symbol $\gg\!=$ to denote monadic substitution. Suppose $x \leqslant_A y$, we show

$$x \gg\!= f \quad \leqslant_A \quad y \gg\!= f.$$

Using the definition of $\leqslant_A$, we must show, for any representation R of (S, A),

$$i_R(x \gg\!= f) \quad \leqslant_R \quad i_R(y \gg\!= f).$$

Since i_R is a morphism of representations, it is compatible with the substitutions of Σ and $\bar{R}$; we have

$$i_R(x \gg\!= f) \quad = \quad i_R(x) \gg\!= i_R \circ f.$$

Rewriting this equality and its equivalent for y in the current goal yields the goal

$$i_R(x) \gg\!= i_R \circ f \quad \leqslant_A \quad i_R(y) \gg\!= i_R \circ f,$$

which is true since the substitution of R (whose underlying map is that of $\bar{R}$) is monotone in the first argument (cf. Remark 61) and $i_R(x) \leqslant_R i_R(y)$ by assumption. We hence have defined a monad Σ_A on Δ. We interrupt the proof for an important lemma:

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

Lemma 58. Given a classic functor $V : \mathsf{Rep}^{\Delta}(S) \to \mathrm{LMod}(\Delta, \mathsf{wPre})$ from the category of representations in monads on Δ to the large category of modules over such modules with codomain category wPre, we have

$$x \leqslant_A y \in V(\Sigma)(X) \quad \Leftrightarrow \quad \forall R : \mathsf{Rep}(S, A), \quad V(i_R)(x) \leqslant_{V_X^R} V(i_R)(y),$$

where now and later we omit the argument X, e.g. in $V(i_R)(X)(x)$.

Proof of Lemma 58. The proof is done by induction on the derivation of 'V classic'. The only interesting case is where $V = M \times N$ is a product:

$$\begin{aligned}(x_1, y_1) \leqslant (x_2, y_2) &\Leftrightarrow x_1 \leqslant x_2 \wedge y_1 \leqslant y_2 \\ &\Leftrightarrow \forall R, M(i_R)(x_1) \leqslant M(i_R)(x_2) \wedge \forall R, N(i_R)(y_1) \leqslant N(i_R)(y_2) \\ &\Leftrightarrow \forall R, M(i_R)(x_1) \leqslant M(i_R)(x_2) \wedge N(i_R)(y_1) \leqslant N(i_R)(y_2) \\ &\Leftrightarrow \forall R, V(i_R)(x_1, y_1) \leqslant V(i_R)(x_2, y_2).\end{aligned}$$

□

— *Representing S in Σ_A.*
Any arity $s \in S$ should be represented by the module morphism s^{Σ}, i.e. the representation of s in Σ. We have to show that those representations are compatible with the preorder A. Given $x \leqslant_A y$ in $\mathrm{dom}(s, \Sigma)(X)$, we show (omitting the argument X in $s^{\Sigma}(X)(x)$)

$$s^{\Sigma}(x) \quad \leqslant_A \quad s^{\Sigma}(y).$$

By definition, we have to show that, for any representation R as before,

$$i_R(s^{\Sigma}(x)) \quad \leqslant_R \quad i_R(s^{\Sigma}(y)).$$

Since i_R is a morphism of representations, it commutes with the representational module morphisms – the corresponding diagram is similar to the diagram of Definition 39. By rewriting with this equality we obtain the goal

$$s^R\Big(\big(\mathrm{dom}(s)(i_R)\big)(x)\Big) \quad \leqslant_R \quad s^R\Big(\big(\mathrm{dom}(s)(i_R)\big)(y)\Big).$$

This goal is proved by instantiating Lemma 58 with the classic S-module $\mathrm{dom}(s)$ (cf. Definition 52) and the fact that s^R is monotone. We hence have established a representation – which we call Σ_A – of S in Σ_A.
— *Σ_A satisfies A.*
The next step is to show that the representation Σ_A satisfies A. Given an inequation

$$\alpha \leqslant \gamma : U \to V$$

of A with a classic S-module V, we must show that for any set X and any $x \in U(\Sigma_A)(X)$ in the domain of α we have

$$\alpha_X^{\Sigma_A}(x) \quad \leqslant_A \quad \gamma_X^{\Sigma_A}(x). \tag{3.2}$$

In the following, we omit the subscript X. By Lemma 58, the goal is equivalent to

$$\forall R : \mathsf{Rep}(S, A), \quad V(i_R)(\alpha^{\Sigma_A}(x)) \quad \leqslant_{V_X^R} \quad V(i_R)(\gamma^{\Sigma_A}(x)). \tag{3.3}$$

Let R be a representation of (S, A). We continue by proving (3.3) for R. By Remark 47 and the fact that i_R is also the carrier of a morphism of S–representations from $\Delta\Sigma$ to R

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

(cf. Lemma 42) we can rewrite the goal as

$$\alpha^R\big(U(i_R)(x)\big) \quad \leqslant_{V_X^R} \quad \gamma^R\big(U(i_R)(x)\big),$$

which is true since R satisfies A.

— *Initiality of* Σ_A.

Given any representation R of (S, A), the morphism i_R is monotone with respect to the orders on Σ_A and R by construction of $\leqslant_A$. It is hence a morphism of representations from Σ_A to R. Unicity of the morphisms i_R follows from its unicity in the category of representations of S, i.e. without inequations. Hence Σ_A is the initial object in the category of representations of (S, A). □

Example 59 (Example 55 continued). The only inequation of the signature $\Lambda\beta$ is classic. The initial representation of $\Lambda\beta$ is given by the monad $\mathrm{LC}\beta$ together with the $\mathrm{LC}\beta$-module morphisms Abs and App (cf. Example 15) as representation structure.

3.4. *Some remarks*

Remark 60 (about 'generating' inequations). Given a 2-signature (S, A) and a representation R of S, the representation morphism of modules s^R of any arity s of S is monotone. For the initial representation Σ_A of (S, A) this means that any reduction between terms of Σ, which is specified by an inequation of A, is automatically propagated into subterms. Similarly, the reduction relation on Σ generated by A is by construction reflexive and transitive, since we consider representations in monads with codomain Pre. For the example of $\Lambda\beta$ this means that in order to obtain the 'complete' reduction relation $\twoheadrightarrow_\beta$, it is sufficient to specify only one inequation for the β-rule

$$\lambda M(N) \;\leqslant\; M[* := N].$$

Remark 61 (about substitution). The substitution in Example 5 is compatible with the order on terms in the following sense:

1. $M \leqslant N$ implies $M[* := A] \leqslant N[* := A]$ and
2. $A \leqslant B$ implies $M[* := A] \leqslant M[* := B]$.

The first implication is a general fact about relative monads on Δ: for any such monad P and any $f \in \mathsf{Pre}(\Delta V, PW)$, the substitution $\sigma_{X,Y}(f) \in \mathsf{Pre}(PV, PW)$ is monotone. The second monotonicity property, however, is *false* for a monad on Δ in general. As an example, consider the monad given by

$$\begin{aligned} F(V) ::= \quad & \mathrm{Var} : V \to F(V) \\ & \mid \bot : F(V) \\ & \mid (\Rightarrow) : F(V) \to F(V) \to F(V), \end{aligned}$$

equipped with a preorder which is contravariant in the first argument of the arrow constructor $(\Rightarrow)$. It is then clear that substituting in this position, the first argument of $(\Rightarrow)$, does in fact *reverse* the order on terms, i.e. we obtain (using $\Rightarrow$ infixed)

$$A < B \quad \text{implies} \quad (* \Rightarrow M)[* := B] < (* \Rightarrow M)[* := A].$$

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

A different definition of monad which would enforce implication 2 to hold – and hence not include the example F – is given by considering Pre as a 2-category enriched over itself: given morphisms $f, g : \mathsf{Pre}(V, W)$ we say that there is precisely one 2-cell

$$f \Rightarrow g \quad \text{iff} \quad f \leqslant g \quad \text{iff} \quad \forall v : V, f(v) \leqslant g(v).$$

A monad P would then have to be equipped with a substitution action that is given, for any two sets V and W, by *a functor* (of preorders)

$$\sigma_{V,W} : \mathsf{Pre}(\Delta V, PW) \to \mathsf{Pre}(PV, PW).$$

Remark 62 (about finite contexts). Altenkirch *et al.* (2010) characterize the untyped lambda calculus as a relative monad on the inclusion functor $i : \mathsf{Fin} \to \mathsf{Set}$ from finite sets to sets. An anonymous referee suggested combining our viewpoint – syntax as monad over $\Delta : \mathsf{Set} \to \mathsf{Pre}$ – with Altenkirch *et al.*'s: one might consider the lambda calculus as a relative monad on the composition $\Delta \circ i : \mathsf{Fin} \to \mathsf{Pre}$, and, more generally, one might consider representations of a signature (S, A) over monads on $\Delta \circ i : \mathsf{Fin} \to \mathsf{Pre}$. The above theorem remains true when replacing monads on Δ by monads on $\Delta \circ i$ everywhere. An equivalence between the theorem thus obtained and our Theorem 57 might be established in a way similar to what Zsidó does in her PhD thesis: she shows, by means of adjunctions between the respective categories of models, the equivalence between the approach of Fiore *et al.* (1999) – based on monoids over finite contexts – and the approach of Hirschowitz and Maggesi (2007a), where models are built from monads on the category Set, i.e. over arbitrary contexts.

Remark 63 (about monads on Pre). As mentioned in Section 1.2, Ghani and Lüth (2003) and Hirschowitz and Maggesi (2010) suggest the use of monads over the category Pre of preordered sets for modelling syntax with a rewriting relation. Indeed, representations of a signature (S, A) could be analogously defined for such monads. The above construction of the initial representation of (S, A) carries over to representations in such monads, thus yielding an initiality result in which syntax is modelled as monad on Pre. It might be interesting to establish a precise connection – e.g. in form of adjunctions – between the resulting categories of representations in monads on Pre and representations in relative monads on Δ.

4. Formalization in the proof assistant Coq

In this section, we explain some elements of the formalization of the initiality theorem in the proof assistant Coq. However, we only explain the implementations of definitions and lemmas that are specific to the theorem. We base ourselves on a general library of category-theoretic concepts, the formalization details of which we do not go into. The interested reader can find an in–depth description and the complete Coq code on the author's web page. The implementation of categories, monads and modules over monads (which are analogous to the implementation of their relative counterparts used here) is explained in detail by Ahrens and Zsidó (2011).

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

For a morphism f from object a to object b in any category we write `f : a ---> b`. Composition of morphisms $f : a \to b$ and $g : b \to c$ is written `f ;; g`.

4.1. *Arities as lists*

According to Definition 32, a 1-signature consists of an indexing type and, for each index, a list of natural numbers, indicating the number of arguments of a constructor, as well as the number of variables bound in each argument. In the formalization they are simply called 'signatures':

```
Notation "[ T ]" := (list T) (at level 5).
Record Signature : Type := {
  sig_index : Type ;
  sig : sig_index -> [nat] }.
```

Next we formalize context extension according to a natural number, cf. Section 2.3. These definitions are important for the definition of the module morphisms we associate to an arity, cf. below. Context extension is actually functorial. Given a natural number `n` and a set of variables `V`, we recursively define the set `V ** n` to be the set `V` enriched with `n` additional variables:

```
Fixpoint pow (n : nat) (V : TYPE) : TYPE :=
  match n with
  | 0 => V
  | S n' => pow n' (option V)
  end.
Notation "V ** n" := (pow n V) (at level 10).
Fixpoint pow_map (l : nat) V W (f : V ---> W) :
        V ** l ---> W ** l :=
  match l return V ** l ---> W ** l with
  | 0 => f
  | S n' => pow_map (^ f)
  end.
Notation "f ^^ l" := (pow_map (l:=l) f) (at level 10).
```

4.2. *Representations*

Given an arity s, i.e. a list of natural numbers s (cf. Definition 32), and a relative monad P on the functor Δ, we need to define the product module P^s. More generally, we define M^s for any P-module M with codomain Pre. We build this module from scratch instead of relying on the category-theoretic constructions such as product and derivation functor for the module categories, allowing us to omit the insertion of isomorphisms in the style of Lemmas 28 and 29. The reasons for this design choice are explained elsewhere (Ahrens and Zsidó 2011). Given any module `M` over a monad `P` from sets to preordered sets, we define the product type `prod_mod_c` as an inductive type family parametrized by a context

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

and dependent on a list of naturals. Actually we define at first the carrier depending not on a module, but just on a carrier map `M : TYPE -> Ord`. The relation on the product is induced by that on `M`.

```
Variable M : TYPE -> Ord.
Inductive prod_mod_c (V : TYPE) : [nat] -> Type :=
  | TTT : prod_mod_c V nil
  | CONSTR : forall b bs,
        M (V ** b)-> prod_mod_c V bs -> prod_mod_c V (b::bs) .
Notation "a -:- b" := (CONSTR a b) (at level 60).
Inductive prod_mod_c_rel (V : TYPE) : forall n, relation (prod_mod_c M V
    n) :=
  | TTT_rel : forall x y : prod_mod_c M V nil, prod_mod_c_rel x y
  | CONSTR_rel : forall n l, forall x y : M (V ** n),
         forall a b : prod_mod_c M V l, x << y ->
     prod_mod_c_rel a b -> prod_mod_c_rel (x -:- a) (y -:- b).
```

Note that the infixed '<<' is overloaded notation and denotes the relation of any preordered set. For any given list `a` of naturals and any set `V` of variables, the set `prod_mod_c V a` equipped with the relation `prod_mod_c_rel V a` is in fact a preordered set. For the proof of transitivity we rely on the Coq tactic `dependent induction`, thus on the axioms

```
JMeq.JMeq_eq : forall (A : Type) (x y : A), x ~= y -> x = y
Eqdep.Eq_rect_eq.eq_rect_eq : forall (U : Type) (p : U)
                              (Q : U -> Type) (x : Q p)
                              (h : p = p), x = eq_rect p Q x p h
```

from the Coq standard library. Now if `M` is not just a map of type `TYPE -> Ord`, but a module over some relative monad `P` over `Delta` (the functor $\Delta : \mathsf{Set} \to \mathsf{Pre}$), we equip the product map with a module substitution in form of a recursive function:

```
Variable M : RModule P Delta.
Fixpoint pm_mkl l V W (f : Delta V ---> P W)
      (X : prod_mod_c (fun V => M V) V l) : prod_mod_c _ W l :=
     match X in prod_mod_c _ _ l return prod_mod_c (fun V => M V) W l
         with
     | TTT => TTT _ W
     | elem -:- elems =>
      rmkleisli (RModule_struct := M) (lshift _ f) elem -:- pm_mkl f
          elems
     end.
(* ... *)
Definition prod_mod (a : [nat]) := Build_RModule (prod_mod_struct a).
```

Afterwards we prove by induction that this map is indeed monotone with respect to the preorder `prod_mod_c_rel` defined above. Altogether, we obtain a module `prod_mod M l` for any module `M : RMOD P Ord` and any list of naturals `l`.

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

To any arity `ar : [nat]` and a module `M` over a monad `P` we associate a type of module morphisms `modhom_from_arity ar M`. Representing `ar` in `M` then means giving a term of type `modhom_from_arity ar M`. In Definition 35, we have defined representations in *monads* only. Indeed we instantiate `M` with the tautological module later.

```
Variable P : RMonad Delta.
Definition modhom_from_arity (M : RModule P Ord) (ar : [nat]) : Type :=
    RModule_Hom (prod_mod M ar) M.
```

For the rest of the section, a representation `S` is fixed via a Coq section variable:

```
Variable S : Signature.
```

As just mentioned, representing the signature `S` in a monad `P` (cf. Definition 35) means providing a suitable module morphism for any arity of `S`, i.e. providing, for any element of the indexing set `sig_index S`, a term of type `modhom_from_arity P (sig i)`:

```
Definition Repr (P : RMonad Delta) :=
  forall i : sig_index S, modhom_from_arity P (sig i).
Record Representation := {
  rep_monad :> RMonad Delta ;
  repr : Repr rep_monad }.
```

Note that the projection `rep_monad` is declared as a *coercion* via the special syntax `:>`. This coercion allows for abuse of notation in Coq as we do informally according to Definition 35, cf. Section 4.6 for a use of this coercion.

4.3. *Morphisms of representations*

A morphism of representations from P to Q is given by a monad morphism $f : P \to Q$ between the underlying monads such that a diagram commutes for any arity, cf. Definition 39. The main task in the implementation is to define this diagram for a given arity ℓ, and, more specifically, the left vertical morphism

$$\mathrm{dom}(\ell, f) = f^{\ell} : P^{\ell} \to f^{*}Q^{\ell},$$

using the notation of Definition 27. Since the carrier of P^{ℓ} is defined as an inductive type, it makes sense to define f^{ℓ} by recursion on the inductive type underlying P^{ℓ}, named `prod_mod_c P V l`:

```
Variables P Q : RMonad Delta.
Variable f : RMonad_Hom P Q.
Fixpoint Prod_mor_c (l : [nat]) (V : TYPE) (X : prod_mod_c (fun V => P V)
     V l) :
                 (prod_mod_c _ V l) :=
  match X in prod_mod_c _ _ l
  return f* (prod_mod Q l) V with
  | TTT => TTT _ _
  | elem -:- elems => f _ elem -:- Prod_mor_c elems
  end.
```

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

Proving this map monotone is an easy exercise, as well as its commutation property with substitution, yielding the aforementioned module morphism.

Now we have all the ingredients we need in order to define the diagram of Definition 39. For an arity a the diagram reads as follows:

```
Variable a : [nat].
Variable RepP : modhom_from_arity P a.
Variable RepQ : modhom_from_arity Q a.
Notation "f * M" := (# (PbRMOD f _ ) M).
Definition commute := Prod_mor a ;; f * RepQ == RepP ;; f^.
```

Here `f^` denotes the module morphism induced by a monad morphism, cf. Definition 21. Using the preceding definition, we define morphisms of representations of `S`:

```
Variables P Q : Representation.
Class Representation_Hom_struct (f : RMonad_Hom P Q) :=
   repr_hom_s : forall i : sig_index S, commute f (repr P i) (repr Q i).
Record Representation_Hom : Type := {
  repr_hom_c :> RMonad_Hom P Q;
  repr_hom :> Representation_Hom_struct repr_hom_c }.
```

4.4. *Category of representations*

In this section, we describe in more detail the category of representations of a 1-signature, cf. Definition 41. The composition of morphisms of representations $f : P \to Q$ and $g : Q \to R$ is essentially done by composing the underlying monad morphisms. One has to show that this morphism indeed commutes with the representation morphisms of P and R. Similarly, the identity monad morphism of (the monad underlying) a representation P yields a morphism of representations. Fed with some suitable lemma, the `Program` framework does the job for us:

```
Program Instance Rep_comp_struct :
     Representation_Hom_struct (RMonad_comp f g).
Program Instance Rep_Id_struct : Representation_Hom_struct (RMonad_id P).
```

Since equality of morphisms of representations is defined as equality of the underlying monad morphisms, the categorical properties of composition are established already as part of the definition of the category `RMONAD F` for any functor `F`.

```
Program Instance REP_struct : Cat_struct (@Representation_Hom S) := {
  mor_oid a c := eq_Rep_oid a c;
  id a := Rep_Id a;
  comp P Q R f g := Rep_Comp f g }.
Definition REP := Build_Cat REP_struct.
```

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

4.5. *Initiality without inequations*

We construct the initial object of the category REP. In the informal proof of Theorem 43 this initial object is the image under a left adjoint of the initial object in a category of representations as defined by Hirschowitz and Maggesi (2007a). For the formal proof we decide to implement the initial object of REP directly, in order to obtain a compact formalization.

```
Inductive UTS (V : TYPE) : TYPE :=
  | Var : V -> UTS V
  | Build : forall (i : sig_index S), UTS_list V (sig i) -> UTS V
with
UTS_list (V : TYPE) : [nat] -> Type :=
  | TT : UTS_list V nil
  | constr : forall b bs,
      UTS (V ** b) -> UTS_list V bs -> UTS_list V (b::bs).
Notation "a -::- b" := (constr a b).
Definition UTS_sm V := Delta (UTS V).
```

We define renaming and, built on top of renaming, substitution:

```
Fixpoint rename (V W: TYPE ) (f : V ---> W) (v : UTS V):=
    match v in UTS _ return UTS W with
    | Var v => Var (f v)
    | Build i l => Build (l //-- f)
    end
with
  list_rename V t (l : UTS_list V t) W (f : V ---> W) : UTS_list W t :=
     match l in UTS_list _ t return UTS_list W t with
     | TT => TT W
     | constr b bs elem elems => elem //- f ^^ b -::- elems //-- f
     end
where "x //- f" := (rename f x)
and "x //-- f" := (list_rename x f).
Fixpoint subst (V W : TYPE) (f : V ---> UTS W) (v : UTS V) :
 UTS W := match v in UTS _ return UTS _ with
    | Var v => f v
    | Build i l => Build (l >>== f)
    end
with
 list_subst V W t (l : UTS_list V t) (f : V ---> UTS W) : UTS_list W t :=
     match l in UTS_list _ t return UTS_list W t with
     | TT => TT W
     | elem -::- elems =>
       elem >== _lshift f -::- elems >>== f
     end
```

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

```
where "x >== f" := (subst f x)
and "x >>== f" := (list_subst x f).
```

Renaming and substitution as just defined correspond to functoriality and monadic substitution for representations in monads as done by Hirschowitz and Maggesi (2007a). According to Lemma 4, we have to apply the functor `Delta` in some places:

```
Program Instance UTS_sm_rmonad : RMonad_struct Delta UTS_sm := {
  rweta c := #Delta (@Var c);
  rkleisli a b f := #Delta (subst f) }.
Canonical Structure UTSM := Build_RMonad UTS_sm_rmonad.
```

The monad `UTSM` is easily equipped with a representation of the signature `S`; the carrier of the representation of `i : sig_index S` is given by the function

```
fun (X : prod_mod_c _ V (sig i)) => Build (i:=i) (UTSl_f_pm (V:=V) X)
```

that is, by the constructor `Build i` of the type `UTS`, precomposed with an isomorphism `UTSl_f_pm` from `prod_mod_c UTS` to `UTS_list`. We thus obtain a representation `UTSRepr` of the signature `S`.

Given another representation, say, `R`, of `S`, the morphism `init` from `UTSRepr` to `R` is defined by recursion:

```
Fixpoint init V (v : UTS V) : R V :=
       match v in UTS _ return R V with
       | Var v => rweta (RMonad_struct := R) V v
       | Build i X => repr R i V (init_list X)
       end
with
 init_list l (V : TYPE) (s : UTS_list V l) : prod_mod R l V :=
    match s in UTS_list _ l return prod_mod R l V with
    | TT => TTT _ _
    | elem -::- elems => init elem -:- init_list elems
    end.
```

This map `init` is compatible with lifting and substitution in `UTSM` and `R`, respectively:

```
Lemma init_lift V x W (f : V ---> W) :
   init (x //- f) = rlift R f (init x).
Lemma init_kleisli V (v : UTS V) W (f : Delta V ---> UTS_sm W) :
  init (v >== f) = rkleisli (f ;; @init_sm W) (init v).
```

where `init_sm W` is the (trivially) monotone version of `init W` – the adjunct of `init W` under the adjunction of Definition 3. The latter of those lemmas constitutes an important part of the proof that `init` is the carrier of a monad morphism from `UTSM` to `R`. It is trivial to prove that `init` is also compatible with the representation structure of `UTSRepr` and `R`,

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

thus the carrier of a morphism of representations called `init_rep : UTSRepr ---> R`. Afterwards unicity of `init_rep` is proved:

```
Lemma init_unique :forall f : UTSRepr ---> R , f == init_rep.
```

Finally we establish initiality by an instance declaration of the corresponding class. The proof field of the class stating unicity is filled automatically by the `Program` framework, using the aforementioned lemma:

```
Program Instance UTS_initial : Initial (REP S) := {
  Init := UTSRepr ;
  InitMor R := init_rep R }.
```

4.6. *Inequations and initial representation of a 2–signature*

For a 1-signature S, an S-module is defined to be a functor from representations of S to the category whose objects are pairs of a monad P and a module M over P, cf. Definition 45. The use of the cumbersome category $\mathrm{LMod}(\Delta, \mathsf{wPre})$ of pairs ensures that representations in a monad P go to P-modules. In Coq we represent this dependency via a dependent type. We do not make use of the functor properties of S-modules.

The below definition makes use of two *coercions*. Firstly, we may write $a : \mathcal{C}$ because the 'object' projection of the category record (whose definition we omit) is declared as a coercion. Secondly, the monad underlying any representation can be accessed without explicit projection using the coercion we mentioned above.

```
Record S_Module := {
  s_mod :> forall R : REP S, RMOD R wOrd;
  s_mod_hom :> forall (R T : REP S)(f : R ---> T),
        s_mod R ---> PbRMod f (s_mod T) }.
Notation "U @ f" := (s_mod_hom U f)(at level 4).
```

Note that we write `U@f` for the image of the morphism of representations `f` under the S-module `U`. Source and target module of `f` are implicit arguments in this application.

A half-equation is a natural transformation between S-modules. We need the naturality condition in the following. Since we have not formalized S-modules as functors, we have to state naturality explicitly:

```
Class half_equation_struct (U V : S_Module)
    (half_eq : forall R : REP S, U R ---> V R) := {
  comm_eq_s : forall (R T : REP S) (f : R ---> T),
      U @ f ;; PbRMod_Hom _ (half_eq T) == half_eq R ;; V @ f }.
Record half_equation (U V : S_Module) := {
  half_eq :> forall R : REP S, U R ---> V R ;
  half_eq_s :> half_equation_struct half_eq }.
```

We now formalize *classic* S-modules. Any list of natural numbers specifies uniquely a classic S-module, cf. Definition 49. Given a list of naturals `codl`, we call this S-module

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

`S_Mod_classic codl`. A *classic half-equation* is a half-equation with a classic codomain, and a classic inequation is a pair of parallel classic half-equations (cf. Definition 56):

```
Definition half_eq_classic (U : S_Module)(codl : [nat]) :=
                            half_equation U (S_Mod_classic codl).
Record ineq_classic := {
  Dom : S_Module;
  Cod : [nat];
  eq1 : half_eq_classic Dom Cod;
  eq2 : half_eq_classic Dom Cod }.
```

Given a representation `P` and a (classic) inequation `e`, we check whether `P` satisfies `e` by pointwise comparison (cf. Definition 54):

```
Definition satisfies_ineq (e : ineq_classic) (P : REP S) :=
  forall c (x : Dom e P c), eq1 _ _ _ x << eq2 _ _ _ x.
Definition Inequations (A : Type) := A -> ineq_classic.
Definition satisfies_ineqs A (T : Inequations A) (R : REP S) :=
     forall a, satisfies_ineq (T a) R.
```

We formalize sets of classic inequations as pairs of an indexing type `A` together with a term of type `Inequations A`, that is, a map from `A` to the type of classic inequations `ineq_classic`. The category of representations of (S, A) is obtained as a full subcategory of the category of representations of S. The following declaration produces a subcategory from predicates on the type of representations and on the (dependent) type of morphisms of representations, yielding the category `INEQ_REP` of representations of (S, A):

```
Variable A : Type.
Variable T : Inequations A.
Program Instance Ineq_Rep : SubCat_compat (REP S)
    (fun P => satisfies_ineqs T P) (fun a b f => True).
Definition INEQ_REP : Cat := SubCat Ineq_Rep.
```

We now construct the initial object of `INEQ_REP`. The relation on the initial object is defined precisely as in the paper proof, cf. Equation (3.1):

```
Definition prop_rel_c X (x y : UTS S X) : Prop :=
     forall R : INEQ_REP, init (FINJ _ R) x << init (FINJ _ R) y.
```

Here, `FINJ _ R` denotes the representation `R` as a representation of `S`, i.e. the injection of `R` in the category `REP S` of representations of `S`. The relation defined above is indeed a preorder, and we define the monad `UTSP` to be the monad whose underlying sets are identical to `UTSM`, namely the sets defined by `UTS`, but equipped with this new preorder. This monad `UTSP` is denoted by Σ_A in the paper proof.

The representation module morphisms of the initial representation `UTSRepr` can be 'reused' after having proved their compatibility with the new order, yielding a representation `UTSProp`. We prove that this representation satisfies the inequations of `T`:

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

```
Lemma UTSPRepr_sig_prop : satisfies_ineqs T UTSProp.
```

We explicitly inject the representation into the category of representations of (S, A):

```
Definition UTSPROP : INEQ_REP :=
 exist (fun R : Representation S => satisfies_ineqs T R) UTSProp
  UTSPRepr_sig_prop.
```

In order to build the initial morphism towards any representation `R : INEQ_REP`, we first build the corresponding morphism in the category of representations of S:

```
Definition init_prop_re : UTSPropr ---> (FINJ _ R) := ...
```

which we then inject, analogously to the initial representation, into the subcategory of representations of (S, A):

```
Definition init_prop : UTSPROP ---> R := exist _ (init_prop_re R) I.
```

Finally, we obtain our Theorem 57. An initial object of a category is given by an object `Init` of this category, a map associating go any object `R` a morphism `InitMor R : Init ---> R`, and a proof of uniqueness of any such morphism. We instantiate the type class `Initial` for the category `INEQ_REP` of representations of (S, A):

```
Program Instance INITIAL_INEQ_REP : Initial INEQ_REP := {
  Init := UTSPROP;
  InitMor := init_prop;
  InitMorUnique := init_prop_unique }.
```

We check its type after closing all the sections, thus abstracting from the section variables:

```
Check INITIAL_INEQ_REP.
INITIAL_INEQ_REP
     : forall (S : Signature) (A : Type) (T : Inequations S A),
       Initial (INEQ_REP (S:=S) (A:=A) T).
```

4.7. *$\Lambda\beta$: Lambda calculus with β reduction*

We implement the example 2-signature $\Lambda\beta$, cf. Example 55. Throughout this section, we use a custom notation in Coq for the datatype of lists:

```
Notation "[[ x ; .. ; y ]]" := (cons x .. (cons y nil) ..).
```

4.7.1. *The 1-signature Λ.* In order to specify the 1-signature Λ (cf. Example 33), we first define an indexing set `Lambda_index` consisting of two elements, `ABS` and `APP`. This indexing set reflects the fact that the signature Λ consists of two arities. The record instance `Lambda` is a term of type `Signature`. The map `sig Lambda` then associates the

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

corresponding lists of naturals to each of these elements, according to Definition 32:

```
Inductive Lambda_index := ABS | APP.
Definition Lambda : Signature := {|
  sig_index := Lambda_index ;
  sig := fun x => match x with
                | ABS => [[ 1 ]]
                | APP => [[ 0 ; 0]]
                end |}.
```

4.7.2. *The* Λ*–inequation* β. The definition of the inequation β (cf. Example 55) is a more challenging task, since a half-equation is not just combinatory data like a 1-arity, but given by suitable module morphisms. At first, we define the substitution of *one* variable (cf. Definition 50) as a half-equation. The carrier `subst_carrier` of the substitution is defined as in Definition 30. Afterwards we prove that this carrier satisfies the properties of a module morphism, that is, is compatible with substitution in the source and target modules. After abstracting from the section variable `R`, we obtain a function `subst_module_mor` which, given any representation `R` of `S`, yields the substitution module morphism associated to (the monad underlying) `R`.

```
Variable S : Signature.
Variable R : REP S.
Definition subst_carrier :
   (forall c : TYPE, (S_Mod_classic_ob [[1; 0]] R) c --->
        (S_Mod_classic_ob [[0]] R) c) := ...
Program Instance sub_struct : RModule_Hom_struct
      (M:=S_Mod_classic_ob [[1 ; 0]] R)
      (N:=S_Mod_classic_ob [[0]] R)
   subst_carrier.
Definition subst_module_mor := Build_RModule_Hom (sub_struct R).
```

The last step is to prove 'naturality'. We recall that we do not implement S-modules as functors, but just as the data part of functors. This is why we put the word *naturality* in quotes. After the proof we define our first half-equation, `subst_half_eq`.

```
Program Instance subst_half_s : half_equation_struct
      (U:= S_Mod_classic [[1 ; 0]])
      (V:= S_Mod_classic [[0]])
   subst_module_mor.
Definition subst_half_eq := Build_half_equation subst_half_s.
```

The definition of the second half-equation of Example 51 is possible for any 1-signature with abstraction and application, such as the 1-signature Λ. To keep the example simple, we only define the half-equation for Λ. The needed steps are precisely the same as for the

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

substitution half-equation, so we just give the statements.

```
Definition beta_carrier:
   (forall c : TYPE, (S_Mod_classic_ob [[1; 0]] R) c --->
        (S_Mod_classic_ob [[0]] R) c) := ...
Program Instance beta_struct : RModule_Hom_struct
      (M:=S_Mod_classic_ob [[1 ; 0]] R)
      (N:=S_Mod_classic_ob [[0]] R)
   beta_carrier.
Definition beta_module_mor := Build_RModule_Hom beta_struct.
Program Instance beta_half_s : half_equation_struct
      (U:=S_Mod_classic Lambda [[1 ; 0]])
      (V:=S_Mod_classic Lambda [[0]])
   beta_module_mor.
Definition beta_half_eq := Build_half_equation beta_half_s.
```

We package both half-equations into one inequation specifying the beta rule of Example 55:

```
Definition beta_rule : ineq_classic Lambda := {|
   eq1 := beta_half_eq ;
   eq2 := subst_half_eq Lambda |}.
```

We can now associate a short name to the category of representations of $\Lambda\beta$, where, for increased clarity, we specify the implicit arguments:

```
Definition Lambda_beta_Cat := INEQ_REP
    (S:=Lambda)(A:=unit)(fun x : unit => beta_rule).
```

Our formal definition allows that an inequation appears multiple times in a 2-signature, whereas in the informal definition we have *sets* of inequations. Unlike for arities, having several copies of the same inequation does not change the resulting category or the initial object. The initial representation is obtained via the specification

```
Definition Lambda_beta := @Init _ _ _
       (INITIAL_INEQ_REP (fun x : unit => beta_rule)).
```

5. Conclusions & future work

We have presented an initiality result for abstract syntax which integrates semantics specified by reduction rules by means of preorders. It is based on relative monads and modules over such monads. Reduction rules are specified by *inequations*, whose definition is largely inspired by the equations of Hirschowitz and Maggesi (2007b). For any 2-signature (S, A) with classic inequations, we construct the initial object in the category of representations of (S, A). The theorem is proved formally in the proof assistant Coq.

On another line of work (Ahrens 2012) we have extended Zsidó's initiality result (Zsidó 2010, Chap. 6) for simply-typed syntax to allow for models over *varying* object

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

types. In this way, initiality accounts for translation between languages over different types. Both lines of work, varying object types and the integration of semantics, can be combined: in a forthcoming work we will present an initiality result which allows for specification of reductions as well as change of object types. As an example, we consider the language PCF with its usual small-step semantics. By equipping LCβ with a representation of PCF, we obtain a translation of PCF to LCβ which is faithful with respect to semantics. Our approach should be extended to more complex type systems featuring dependent types or polymorphism. Another interesting feature to work on are *conditional* reductions/rewritings. A more fine-grained account of reduction might be given by considering *graphs* instead of preordered sets.

Acknowledgements

We are indebted to André Hirschowitz and Marco Maggesi for invaluable discussions. We thank the anonymous referees for their careful reading and suggestions. We finally thank Thorsten Altenkirch and Conor McBride for their editorial work.

References

Ahrens, B. (2012) Extended initiality for typed abstract syntax. *Logical Methods in Computer Science* **8.2** 1–35. (doi: 10.2168/LMCS-8(2:1)2012.)

Ahrens, B. and Zsidó, J. (2011) Initial semantics for higher–order typed syntax in Coq. *Journal of Formalized Reasoning* **4.1** 25–69.

Altenkirch, T. and Reus, B. (1999) Monadic presentations of lambda terms using generalized inductive types. In: *Computer Science Logic, 13th International Workshop, CSL '99* 453–468.

Altenkirch, T. *et al.* (2010) Monads need not be endofunctors. In: Luke Ong, C.-H. (ed.) FOSSACS. *Springer Lecture Notes in Computer Science* **6014** 297–311. ISBN: 978-3-642-12031-2.

Barendregt, H. and Barendsen, E. (1994) Introduction to lambda calculus. Available at: ftp://ftp.cs.ru.nl/pub/CompMath.Found/lambda.pdf. revised 2000.

Bird, R. S. and Meertens, L. (1998) Nested datatypes. In: Jeuring, J. (ed.) Proceedings of Mathematics of Program Construction. *Springer Lecture Notes in Computer Science* **1422** 52–67.

Coq (2010) The coq proof assistant. Available at: http://coq.inria.fr. URL: http://coq.inria.fr.

Fernández, M. and Gabbay, M. J. (2007) Nominal rewriting. *Information and Computation* **205.6** 917–965. doi: 10.1016/j.ic.2006.12.002.

Fiore, M. (2002) Semantic analysis of normalisation by evaluation for typed lambda calculus. In: *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming. PPDP'02*, Pittsburgh, PA, USA: ACM 26–37. ISBN: 1-58113-528-9. doi: http://doi.acm.org/10.1145/571157.571161.

Fiore, M. (2005) Mathematical models of computational and combinatorial structures. In: Sassone, V. (ed.) FoSSaCS. *Springer Lecture Notes in Computer Science* **3441** 25–46. ISBN: 3-540-25388-2.

Fiore, M. and Hur, C.-K. (2007) Equational systems and free constructions (extended abstract). In: Arge, L. *et al.* (ed.) ICALP. *Springer Lecture Notes in Computer Science* **4596** 607–618. ISBN: 978-3-540-73419-2.

Fiore, M. *et al.* (1999) Abstract syntax and variable binding. In: *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science. LICS '99,* Washington, DC, USA: IEEE Computer Society 193–202. ISBN: 0-7695-0158-3.

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press

Gabbay, M. and Pitts, A. (1999) A new approach to abstract syntax involving binders. In: *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science. LICS '99*, Washington, DC, USA: IEEE Computer Society 214–224. ISBN: 0-7695-0158-3.

Gabbay, M. J. and Pitts, A. M. (2001) A new approach to abstract syntax with variable binding. *Formal Aspects of Computing* **13.3–5** 341–363. doi: http://doi.ieeecomputersociety.org/10.1109/LICS.1999.782617.

Ghani, N. and Lüth, C. (2003) Rewriting via coinserters. *Nordic Journal of Computing* **10.4** 290–312.

Hirschowitz, A. and Maggesi, M. (2007a) Modules over monads and linearity. In: Leivant, D. and de Queiroz, R. J. G. B. (eds.) WoLLIC. *Springer Lecture Notes in Computer Science* **4576** 218–237. ISBN: 978-3-540-73443-7.

Hirschowitz, A. and Maggesi, M. (2007b) The algebraicity of the lambda-calculus. In: CoRR abs/0704.2900. Available at http://arxiv.org/abs/0704.2900.

Hirschowitz, A. and Maggesi, M. (2010) Modules over monads and initial semantics. *Information and Computation* **208.5** 545–564.

Hirschowitz, T. (2013) Cartesian closed 2-categories and permutation equivalence in higher-order rewriting. *Logical Methods in Computer Science* **9** (3). Available at: http://dx.doi.org/10.2168/LMCS9(3:10)2013.

Hofmann, M. (1999) Semantical analysis of higher-order syntax. In: *14th Annual Symposium on Logic in Computer Science*, IEEE Computer Society Press 204–213.

Hur, C.-K. (2010) *Categorical Equational Systems: Algebraic Models and Equational Reasoning*, Ph.D. thesis, University of Cambridge, UK.

Mac Lane, S. (1998) *Categories for the Working Mathematician,* Second. Graduate Texts in Mathematics volume 5, New York: Springer-Verlag xii+314. ISBN: 0-387- 98403-8.

Miculan, M. and Scagnetto, I. (2003) A framework for typed HOAS and semantics. In: *Principles and Practice of Declarative Programming*, ACM 184–194. ISBN: 1-58113-705-2.

Pitts, A. M. (2003) Nominal logic, a first order theory of names and binding. *Information and Computation* **186** 165–193.

Power, J. (2007) Abstract syntax: Substitution and binders. *Electronic Notes in Theoretical Computer Science* **173** 3–16. ISSN: 1571-0661. doi: 10.1016/j.entcs.2007.02.024.

Tanaka, M. and Power, J. (2005) A unified category-theoretic formulation of typed binding signatures. In: *Proceedings of the 3rd ACM SIGPLAN Workshop on Mechanized Reasoning about Languages with Variable Binding. MERLIN '05*, Tallinn, Estonia: ACM, 13–24. ISBN: 1-59593-072-8. doi: http://doi.acm.org/10.1145/1088454.1088457.

Zsidó, J. (2010) Typed abstract syntax. Available at http://tel.archives-ouvertes.fr/tel- 00535944/. Ph.D. thesis, University of Nice, France.

https://doi.org/10.1017/S0960129514000103 Published online by Cambridge University Press