Initial semantics for reduction rules

Ahrens, Benedikt

DOI:
10.23638/LMCS-15(1:28)2019

Document Version
Peer reviewed version

Citation for published version (Harvard):
Ahrens, B 2019, 'Initial semantics for reduction rules', *Logical Methods in Computer Science*, vol. 15, no. 1, pp. 28:1-28:45. https://doi.org/10.23638/LMCS-15(1:28)2019

INITIAL SEMANTICS FOR REDUCTION RULES

BENEDIKT AHRENS

Abstract. We give an algebraic characterization of the syntax and operational semantics of a class of simply–typed languages, such as the language PCF: we characterize simply–typed syntax with variable binding and equipped with reduction rules via a universal property, namely as the initial object of some category of models. For this purpose, we employ techniques developed in two previous works: in the first work we model syntactic translations between languages *over different sets of types* as initial morphisms in a category of models. In the second work we characterize untyped syntax *with reduction rules* as initial object in a category of models. In the present work, we combine the techniques used earlier in order to characterize simply–typed syntax with reduction rules as initial object in a category. The universal property yields an operator which allows to specify translations—that are semantically faithful by construction—between languages over possibly different sets of types.

As an example, we upgrade a translation from PCF to the untyped lambda calculus, given in previous work, to account for reduction in the source and target. Specifically, we specify a reduction semantics in the source and target language through suitable rules. By equipping the untyped lambda calculus with the structure of a model of PCF, initiality yields a translation from PCF to the lambda calculus, that is faithful with respect to the reduction semantics specified by the rules.

This paper is an extended version of an article published in the proceedings of WoLLIC 2012.

1. Introduction

The syntactic structure of languages is captured algebraically by the notion of *initial algebra*: Birkhoff [Bir35] first characterizes *first-order* syntax with equalities between terms as initial object in some category (without mention of the word "category", of course). Algebraic characterizations of *higher–order* syntax, i.e., of syntax with binding of variables, were given by several people independently in the late 1990's, using different approaches to the representation of binding. Some of these approaches were generalized later to deal with *simple typing* as well as with *equations* between terms, such as β- and η-conversion for functions.

In this work we give an algebraic characterization of simply–typed languages equipped with *directed* equations. Directed equations provide a more faithful model for computation—or *reduction*—than equations and emphasize the *dynamic* point of view of reductions rather than the static one given by equations, as pointed out in [GTL89].

The languages we consider in this work are *purely functional*; in particular, we do not treat memory, environment, effects or exceptions. The rewrite rules we consider are

Preprint submitted to
Logical Methods in Computer Science

unconditional rewrite rules, such as β- and η-reduction. We do not treat conditional rewrite rules as needed, e.g., in the definition of bisimulation for process calculi.

By including reduction rules into our treatment of the syntax of programming languages, we obtain a mechanism—a *recursion operator*—that allows us to define translations between languages that are *semantically faithful* by construction. A semantically faithful translation f is a translation that commutes with reduction: it maps two terms t and t', where t reduces to t', to terms $f(t)$ and $f(t')$ such that $f(t)$ reduces to $f(t')$ in the target language—an important property of any *reasonable* translation between programming languages.

Our definitions ensure furthermore that translations specified via this operator are by construction compatible with substitution (through our use of (relative) monads) and typing, in addition to reduction.

As an example, we specify a translation from PCF to the untyped lambda calculus ULC using this category–theoretic iteration operator. This translation is by construction faithful with respect to reduction in PCF and ULC. This example is verified formally in the proof assistant Coq [CDT12]. The Coq files as well as documentation are available online[1].

1.1. **Difference to the conference version of this article.** The present work is an extended version of a previously published work [Ahr12b]. In that previous work, the main theorem [Ahr12b, Thm. 44] is stated, but no proof is given. In the present work, we review the definitions given in the earlier work and present a proof of the main theorem. Afterwards, we explain in detail the formal verification in the proof assistant Coq [CDT12] of an instance of this theorem, for the simply–typed programming language PCF. Finally, we illustrate the iteration operator coming from initiality by specifying an executable certified translation in Coq from PCF to the untyped lambda calculus.

1.2. **Summary.** We define a notion of *2–signature* which allows the specification of the *types and terms* of a programming language, as well as its *operational semantics* in form of reduction rules for terms. Specifically, such a 2–signature consists, first, of a 1–signature, say, (S, Σ), where S is a signature for types, and Σ is a signature for terms over the types specified by S. Second, on such a 1–signature (S, Σ), reductions rules are specified through a set A of *directed equations*—also called *reduction rules* or *rules* for short—over (S, Σ). To any 1–signature (S, Σ) we associate a category of *models of* (S, Σ). Given a 2–signature $((S, \Sigma), A)$, the rules of A give rise to a *satisfaction* predicate on the models of (S, Σ), and thus specify the full subcategory of models of (S, Σ) that satisfy the rules of A. We call this subcategory the *category of models of* $((S, \Sigma), A)$. Our main theorem states that this category has an initial object—the programming language associated to $((S, \Sigma), A)$—, which integrates the types and terms generated by (S, Σ), equipped with the reduction relation generated by the rules of A.

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. As will be explained in Section 1.2.1, this characterization is actually a consequence of an earlier result [Ahr12a].

[1] `https://github.com/benediktahrens/monads`

(2) Afterwards we consider *directed equations* specifying *reduction rules*. Given a family of reduction rules for terms, we build up on the preceding result to give an algebraic characterization of syntax *with reduction*. Directed equations for *untyped* syntax are considered in earlier work [Ahr16]; in the present work, the main result of that earlier work is carried over to *simply–typed* syntax.

In summary, the merit of this work is to give an algebraic characterization of simply–typed syntax *with reduction rules*, building up on such a characterization for *pure* syntax given earlier [Ahr12a]. Our approach is based on relative monads as defined in [ACU15] from the category Set of sets to the category Pre of preordered sets. Compared to traditional monads, relative monads allow for different categories as domain and codomain.

We now explain the above two points in more detail:

1.2.1. *Pure Syntax.* A 1–signature (S, Σ) is a pair which specifies the types and terms of a language, respectively. Furthermore, it associates a type to any term. To any 1–signature (S, Σ) we associate a category $\mathrm{Model}^{\Delta}(S, \Sigma)$ of *models* of (S, Σ), where a model of (S, Σ) is a pair of a model T of the types specified by S and a model of Σ over T. Models of Σ are relative monads, from (families of) sets to (families of) preordered sets, equipped with some extra structure.

This category has an initial object (cf. Lemma 3.25), which integrates the types and terms freely generated by (S, Σ). We call this object the *(pure) syntax associated to* (S, Σ). 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 in another work [Ahr12a]: in that work, we associate, to any signature (S, Σ), a category $\mathrm{Model}(S, \Sigma)$ of models of (S, Σ), where a model is built from a traditional monad on (families of) sets instead of a relative monad as above. We connect the corresponding categories by exhibiting a pair of adjoint functors (cf. Lemma 3.25) between the category $\mathrm{Model}^{\Delta}(S, \Sigma)$ of models of (S, Σ) and the category $\mathrm{Model}(S, \Sigma)$,

$$\mathrm{Model}(S,\Sigma) \underset{U_*}{\overset{\Delta_*}{\underset{\longleftarrow}{\overset{\longrightarrow}{\bot}}}} \mathrm{Model}^{\Delta}(S,\Sigma) \quad .$$

We thus obtain an initial object in our category $\mathrm{Model}^{\Delta}(S, \Sigma)$ using the fact that left adjoints are cocontinuous: the image under the functor $\Delta_* : \mathrm{Model}(S, \Sigma) \to \mathrm{Model}^{\Delta}(S, \Sigma)$ of the initial object in the category $\mathrm{Model}(S, \Sigma)$ is initial in $\mathrm{Model}^{\Delta}(S, \Sigma)$.

1.2.2. *Syntax with Reduction Rules.* Given a 1–signature (S, Σ), an (S, Σ)-*rule* $e = (\alpha, \gamma)$ associates, to any model R of (S, Σ), a pair (α^R, γ^R) of *parallel* morphisms in a suitable category. In a sense made precise later, we can ask whether

$$\alpha^R \ \leq \ \gamma^R \ ,$$

due to our use of relative monads towards families of *preordered* sets. If this is the case, we say that R *satisfies* the rule e. A *2–signature* is a pair $((S, \Sigma), A)$ consisting of a 1–signature (S, Σ), which specifies the types and terms of a language, together with a family A of

(S, Σ)-rules, which specifies reduction rules on those terms. Given a 2–signature $((S, \Sigma), A)$, we call a *model of* $((S, \Sigma), A)$ any model of (S, Σ) that satisfies each rule of A. The *category of models of* $((S, \Sigma), A)$ is defined to be the full subcategory of models of (S, Σ) whose objects are models of $((S, \Sigma), A)$.

We would like to exhibit an initial object in the category of models of $((S, \Sigma), A)$, and thus must filter out rules which are never satisfied. We call *elementary* (S, Σ)-rule any (S, Σ)-rule whose codomain is of a particular form. Our main result states that for any family A of elementary (S, Σ)-rules, the category of models of $((S, \Sigma), A)$ has an initial object. The class of elementary rules is large enough to account for the fundamental reduction rules; in particular, β- and η-reductions are given by elementary rules. However, our notion of rule does only capture *axioms*, that is, unconditional rewrite rules.

Our definitions ensure that any reduction rule between terms that is expressed by a directed equation $e \in A$ is automatically propagated into subterms in the initial model. The family A of rules hence only needs to contain some "generating" rules, a fact that is well illustrated by the example 2–signature $\Lambda\beta$ of the untyped lambda calculus with β-reduction [Ahr16]: this signature has only one directed equation β which expresses β-reduction at the root of a term,

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

The initial model of $\Lambda\beta$ is given by the untyped lambda calculus, equipped with the reflexive and transitive β-reduction relation $\twoheadrightarrow_\beta$ as presented in [BB94].

1.3. **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 *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 further below.

1.3.1. *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 representation of variable binding:

The *nominal approach* by Gabbay and Pitts [GP99] (see also [GP01, Pit03]) 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 [Hof99] proves an initiality result modelling variable binding in a Higher–Order Abstract Syntax (HOAS) style. Fiore, Plotkin, and Turi [FPT99] (see also subsequent work by Fiore [Fio02, Fio05]) model variable binding through nested datatypes as introduced in [BM98]. The approach of [FPT99] is extended to *simply–typed syntax* in [MS03]. In [TP05] the authors 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 in [Pow07]. In [HM07b] the authors prove an initiality result for untyped syntax based on the notion of *module over a monad*. Their work is extended to simply–typed syntax in [Zsi10].

1.3.2. *Incorporating Semantics.* Rewriting in nominal settings has been examined in [FG07]. In [GL03] the authors 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 of sets resp. Pre of preordered sets. Fiore and Hur [FH07] have extended Fiore's work to integrate semantic aspects into initiality results. In particular, Hur's thesis [Hur10] is dedicated to *equational* systems for syntax with variable binding. In a "Further research" section [Hur10, Chap. 9.3], Hur suggests the use of relations to model "rewrite systems" with *directed* equations. Hirschowitz and Maggesi [HM07b] prove initiality of the set of lambda terms modulo β- and η-conversion in a category of *exponential monads*. In an unpublished paper [HM07a], the authors 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 *directed* equation rather than equation. In a "Future Work" section [HM10, Sect. 8], they mention the idea of using preorders as an approach to model semantics, and they suggest interpreting the untyped lambda calculus with β- and η-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 β-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. Definition 2.6) instead of traditional monads (cf. also [Ahr16]). Relative monads were introduced in [ACU15]. In that work, the authors characterize the untyped lambda calculus as a relative monad over the inclusion functor from finite sets to sets (see Example 2.3). T. Hirschowitz [Hir13] defines a category Sig of 2–signatures for *simply–typed* syntax with reduction rules, and constructs an adjunction between Sig and the category $\mathsf{2CCCat}$ 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 2–cells between terms. His approach differs from ours in the way in which variable binding is modelled: Hirschowitz encodes binding in a Higher–Order Abstract Syntax (HOAS) style through exponentials.

1.4. **Synopsis.** In Section 2 we review the definition of relative monads and modules over such monads as well as their morphisms. Some constructions on monads and modules are given, which will be of importance in what follows.

In Section 3 we define arities, half–equations and directed equations, as well as their models. Afterwards we prove our main result.

In Section 4 we describe the formalization in the proof assistant Coq of an instance of our main result, for the particular case of the language PCF.

Some conclusions and future work are stated in Section 5.

2. Relative monads and modules thereon

Monads capture the notion of *simultaneous substitution* and its properties. This was first observed in [AR99], who characterize the lambda calculus and substitution on it as a monad on the category of sets.

The functor underlying a monad is necessarily endo—this is enforced by the type of monadic multiplication. *Relative monads* were introduced in [ACU15] to overcome this restriction. One of their motivations was to consider the untyped lambda calculus over *finite* contexts and a suitable substitution operation on it as a monad–like structure—in the spirit of aforementioned earlier work [AR99], but now with different categories for source and target.

We review the definition of relative monads and define suitable *colax morphisms of relative monads*. Afterwards we define *modules over relative monads* and generalize the constructions on modules over monads defined in [HM07b] to modules over *relative* monads.

2.1. **Definitions.** We review the definition of relative monad as given in [ACU15] and define suitable morphisms for them. As an example we consider the lambda calculus with β-reduction as a relative monad from sets to preorders, on the functor $\Delta : \mathsf{Set} \to \mathsf{Pre}$ (cf. Definition 2.6). Afterwards we define *modules over relative monads* and carry over the constructions on modules over regular monads of [HM07b] to modules over relative monads.

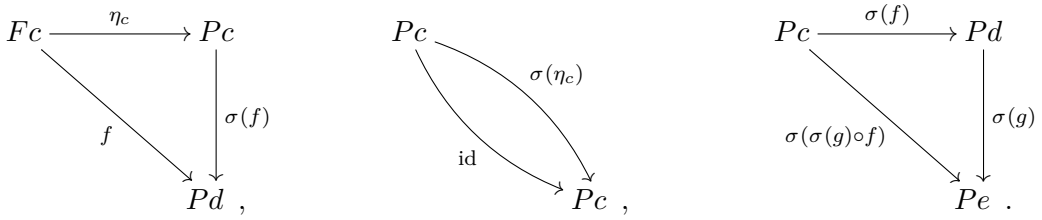
The definition of relative monads is analogous to that of monads *in Kleisli form*, except that the underlying map of objects is between *different* categories. Thus, for the operations to remain well–typed, one needs an additional "mediating" functor, in the following usually called F, which is inserted wherever necessary:

Definition 2.1 (Relative monad, [ACU15])**.** 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 the following data:

- a map $P\colon \mathcal{C} \to \mathcal{D}$ on the objects of $\mathcal{C}$,
- for each object c of $\mathcal{C}$, a morphism $\eta_c \in \mathcal{D}(Fc, Pc)$ and
- for each two objects c, d of $\mathcal{C}$, a *substitution* map (whose subscripts we usually omit)

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

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



We sometimes omit the adjective "relative" when it is clear from the context that the monad referred to is a relative monad.

Remark 2.2. Relative monads on the identity functor $\mathrm{Id} : \mathcal{C} \to \mathcal{C}$ precisely correspond to monads.

Various examples of relative monads are given in [ACU15]. They give one example related to syntax and substitution:

Example 2.3 (Lambda calculus over finite contexts)**.** In [ACU15], the authors consider the untyped lambda calculus as a relative monad on the functor $J : \mathsf{Fin}_{\mathrm{skel}} \to \mathsf{Set}$. Here the category $\mathsf{Fin}_{\mathrm{skel}}$ is the category of finite cardinals, i.e. the skeleton of the category Fin of *finite* sets and maps between finite sets. The category Set is the category of sets, cf. Definition 2.4.

We will give another example (cf. Example 2.9) of how to view syntax *with reduction rules* as a relative monad. For this, we need some more definitions.

Definition 2.4. The category Set is the category of sets and total maps between them, together with the usual composition of maps.

Definition 2.5. The category Pre of preorders has, as objects, sets equipped with a preorder, and, as morphisms between any two preordered sets A and B, the monotone functions from A to B. The category Pre is cartesian closed: given $f, g \in \mathsf{Pre}(A, B)$, we say that $f \leq g$ if and only if for any $a \in A$, $f(a) \leq g(a)$ in B.

Definition 2.6 (Functor $\Delta : \mathsf{Set} \to \mathsf{Pre}$ and forgetful functor)**.** We call $\Delta : \mathsf{Set} \to \mathsf{Pre}$ the left adjoint of the forgetful functor $U : \mathsf{Pre} \to \mathsf{Set}$,

$$\mathsf{Set} \underset{U}{\overset{\Delta}{\underset{\longleftarrow}{\overset{\longrightarrow}{\bot}}}} \mathsf{Pre} \quad .$$

The functor Δ associates, to each set X, the set itself together with the smallest preorder, i.e. the diagonal of X,

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

In other words, for any $x, y \in X$ we have $x \delta_X y$ if and only if $x = y$. The functor $\Delta : \mathsf{Set} \to \mathsf{Pre}$ is a *full embedding*, i.e. it is fully faithful and injective on objects. We have $U \circ \Delta = \mathrm{Id}_{\mathsf{Set}}$. Altogether, 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.

Definition 2.7 (Category of families)**.** Let $\mathcal{C}$ be a category and T be a set, i.e., a discrete category. We denote by $\mathcal{C}^T$ the functor category, an object of which is a T–indexed family of objects of $\mathcal{C}$. Given two families V and W, a morphism $f : V \to W$ is a family of morphisms in $\mathcal{C}$,

$$f : t \mapsto f(t) : V(t) \to W(t) \ .$$

We write $V_t := V(t)$ for objects and morphisms. Given another category $\mathcal{D}$ and a functor $F : \mathcal{C} \to \mathcal{D}$, we denote by $F^T : \mathcal{C}^T \to \mathcal{D}^T$ the functor given by postcomposition with F, that is, $f \mapsto F \circ f$.

Remark 2.8. Note that an adjunction $L \dashv R : \mathcal{C} \to \mathcal{D}$ carries over to functor categories $L^{\mathcal{X}} \dashv R^{\mathcal{X}} : \mathcal{C}^{\mathcal{X}} \to \mathcal{D}^{\mathcal{X}}$.

We use this fact in the following case: given a set T, the adjunction of Definition 2.6 induces an adjunction

$$\mathsf{Set}^T \underset{U^T}{\overset{\Delta^T}{\underset{\longleftarrow}{\overset{\longrightarrow}{\bot}}}} \mathsf{Pre}^T \quad .$$

Example 2.9 (Simply–typed lambda calculus as relative monad on Δ^T)**.** In this example, we consider $\Delta^T : \mathsf{Set}^T \to \mathsf{Pre}^T$. In the Coq code shown here, Type plays the role of Set. Let

$$T := T_{\mathsf{TLC}} ::= \ * \ \mid \ T_{\mathsf{TLC}} \rightsquigarrow T_{\mathsf{TLC}}$$

be the set of types of the simply–typed lambda calculus. Consider the set family of simply–typed lambda terms over T_{TLC}, indexed by typed contexts:

```
Inductive TLC (V : T → Type) : T → Type :=
  | Var : ∀ t, V t → TLC V t
  | Abs : ∀ s t TLC (V + s) t → TLC V (s ~> t)
  | App : ∀ s t, TLC V (s ~> t) → TLC V s → TLC V t.
```

Here the context `V + s` is the context `V` extended by a fresh variable of type `s`—the variable that is bound by the constructor `Abs` (cf. also Section 2.3). We leave the object type arguments implicit and write λM and MN for $\mathrm{Abs}\, M$ and $\mathrm{App}\, MN$, respectively. We equip each set $\mathsf{TLC}(V)(t)$ of lambda terms over context V of object type t with a preorder taken as the reflexive–transitive closure of the relation generated by the rule

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

and its propagation into subterms. This defines a monad TLC_β from families of sets to families of preorders over the functor Δ^T,

$$\mathsf{TLC}_\beta : \mathsf{Set}^T \xrightarrow{\Delta^T} \mathsf{Pre}^T.$$

The family η^{TLC} is given by the constructor Var, and the substitution map

$$\sigma_{X,Y} : \mathsf{Pre}^T(\Delta^T(X), \mathsf{TLC}_\beta(Y)) \to \mathsf{Pre}^T(\mathsf{TLC}_\beta(X), \mathsf{TLC}_\beta(Y)) \tag{2.1}$$

is given by capture–avoiding simultaneous substitution. Via the adjunction of Remark 2.8 the substitution can also be read as

$$\sigma_{X,Y} : \mathsf{Set}^T(X, \mathsf{TLC}(Y)) \to \mathsf{Pre}^T(\mathsf{TLC}_\beta(X), \mathsf{TLC}_\beta(Y)) \ .$$

In the previous example, the substitution of the lambda calculus satisfies an additional monotonicity property: the map $\sigma_{X,Y}$ in Display (2.1) is monotone for the preorders on hom–sets defined in Definition 2.5. This motivates the following definition:

Definition 2.10. Let P be a relative monad on Δ^T for some set T. We say that P is *a reduction monad* if, for any X and Y, the substitution $\sigma_{X,Y}$ is *monotone* for the preorders on $\mathsf{Pre}^T(\Delta^T X, PY)$ and $\mathsf{Pre}^T(PX, PY)$.

The monad TLC_β of Example 2.9 is thus a reduction monad.

Remark 2.11 (Relative monads are functorial)**.** Given a monad P over $F : \mathcal{C} \to \mathcal{D}$, a functorial action for P is defined by setting, for any morphism $f : c \to d$ in $\mathcal{C}$,

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

The functor axioms are easily proved from the monadic axioms.

The substitution $\sigma = (\sigma_{c,d})_{c,d \in |\mathcal{C}|}$ of a relative monad P is binatural:

Remark 2.12 (Naturality of substitution)**.** Given a relative monad P over $F : \mathcal{C} \to \mathcal{D}$, then its substitution σ is natural in c and d. We write $f^*(h) := h \circ f$. For naturality in c we check that the diagram

$$\begin{array}{ccc} c & \mathcal{D}(Fc, Pd) & \xrightarrow{\sigma_{c,d}} \mathcal{D}(Pc, Pd) \\ {\scriptstyle f}\downarrow & {\scriptstyle (Ff)^*}\uparrow & \uparrow{\scriptstyle (Pf)^*} \\ c' & \mathcal{D}(Fc', Pd) & \xrightarrow[\sigma_{c',d}]{} \mathcal{D}(Pc', Pd) \end{array}$$

commutes. Given $g \in \mathcal{D}(Fc', Pd)$, we have

$$\begin{aligned} \sigma(g) \circ Pf &= \sigma(g) \circ \sigma(\eta_{c'} \circ Ff) \\ &\overset{3}{=} \sigma(\sigma(g) \circ \eta_{c'} \circ Ff) \\ &\overset{1}{=} \sigma(g \circ Ff) \ , \end{aligned}$$

where the numbers correspond to the diagrams of Definition 2.1 used to rewrite in the respective step. Similarly we check naturality in d. Writing $h_*(g) := h \circ g$, the diagram

$$\begin{array}{ccc} d & \mathcal{D}(Fc, Pd) & \xrightarrow{\sigma_{c,d}} \mathcal{D}(Pc, Pd) \\ {\scriptstyle h}\downarrow & {\scriptstyle (Ph)_*}\downarrow & \downarrow{\scriptstyle (Ph)_*} \\ d' & \mathcal{D}(Fc', Pd) & \xrightarrow[\sigma_{c',d}]{} \mathcal{D}(Pc', Pd) \end{array}$$

commutes: given $g \in \mathcal{D}(Fc, Pd)$, we have

$$\begin{aligned} Ph \circ \sigma(g) &= \sigma(\eta_{d'} \circ Fh) \circ \sigma(g) \\ &\overset{3}{=} \sigma(\sigma(\eta_{d'} \circ Fh) \circ g) \\ &= \sigma(Ph \circ g) \ . \end{aligned}$$

If $(T_i)_{i \in I}$ is a family of sets and $f : I \to J$ a map of sets, then we obtain a family of sets $(T'_j)_{j \in J}$ by setting $T'_j := \coprod_{\{i | f(i) = j\}} T_i$. The following construction generalizes this reparametrization:

Definition 2.13 (Retyping functor)**.** Let T and T' be sets and $g : T \to T'$ be a map. Let $\mathcal{C}$ be a cocomplete category. The map g induces a functor

$$g^* : \mathcal{C}^{T'} \to \mathcal{C}^T \ , \quad W \mapsto W \circ g \ .$$

The *retyping functor associated to* $g : T \to T'$,

$$\vec{g} : \mathcal{C}^T \to \mathcal{C}^{T'} \ ,$$

is defined as the left Kan extension operation along g, that is, we have an adjunction

$$\mathcal{C}^T \underset{g^*}{\overset{\vec{g}}{\underset{\uparrow}{\rightleftarrows}}} \mathcal{C}^{T'} \quad . \tag{2.2}$$

Put differently, the map $g : T \to T'$ induces an endofunctor $\bar{g}$ on $\mathcal{C}^T$ with object map

$$\bar{g}(V) := \vec{g}(V) \circ g$$

and we have a natural transformation ctype, the unit of the adjunction of Display (2.2),

$$\mathsf{ctype} : \mathrm{Id} \Longrightarrow \bar{g} : \mathcal{C}^T \to \mathcal{C}^T \ .$$

Definition 2.14 (Pointed index sets)**.** Given a category $\mathcal{C}$, a set T and a natural number $n \in \mathbb{N}$, we denote by $\mathcal{C}_n^T$ the category with, as objects, diagrams of the form

$$n \xrightarrow{\mathbf{t}} T \xrightarrow{V} \mathcal{C} \ ,$$

written $(V, t_1, \dots, t_n)$ with $t_i := \mathbf{t}(i)$. A morphism h to another such $(W, \mathbf{t})$ with the same pointing map $\mathbf{t}$ is given by a morphism $h : V \to W$ in $\mathcal{C}^T$; there are no morphisms between objects with different pointing maps. Any functor $F : \mathcal{C}^T \to \mathcal{D}^T$ extends to $F_n : \mathcal{C}_n^T \to \mathcal{D}_n^T$ via

$$F_n(V, t_1, \dots, t_n) := (FV, t_1, \dots, t_n) \ .$$

Remark 2.15. The category $\mathcal{C}_n^T$ consists of T^n copies of $\mathcal{C}^T$, which do not interact. Due to the "markers" $(t_1, \dots, t_n)$ we can act differently on each copy, cf., e.g., Definitions 2.38 and 2.39.

Retyping functors generalize to categories with pointed indexing sets; when changing types according to a map of types $g : T \to T'$, the markers must be adapted as well:

Definition 2.16. Given a map of sets $g : T \to T'$, by postcomposing the pointing map with g, the retyping functor generalizes to the functor

$$\vec{g}(n) : \mathcal{C}_n^T \to \mathcal{C}_n^{T'} \quad , \quad (V, \mathbf{t}) \mapsto (\vec{g}V, g_*(\mathbf{t})) \ ,$$

where $g_*(\mathbf{t}) := g \circ \mathbf{t} : n \to T'$.

Finally there is also a category where families of objects of $\mathcal{C}$ over different indexing sets are mixed together:

Definition 2.17. Given a category $\mathcal{C}$, we denote by $\mathcal{TC}$ the category where an object is a pair (T, V) of a set T and a family $V \in \mathcal{C}^T$ of objects of $\mathcal{C}$ indexed by T. A morphism (g, h) to another such (T', W) is given by a map $g : T \to T'$ and a morphism $h : V \to W \circ g$ in $\mathcal{C}^T$, that is, a family of morphisms in $\mathcal{C}$, indexed by T,

$$h_t : V_t \to W_{g(t)} \ .$$

Suppose $\mathcal{C}$ has an initial object, denoted by $0_{\mathcal{C}}$. Given $n \in \mathbb{N}$, we call $\hat{n} = (n, k \mapsto 0_{\mathcal{C}})$ the object of $\mathcal{TC}$ that associates to any $1 \leq k \leq n$ the initial object of $\mathcal{C}$. We call $\mathcal{TC}_n$ the slice category $\hat{n} \downarrow \mathcal{TC}$. An object of this category consists of an object $(T, V) \in \mathcal{TC}$ whose indexing set "of types" T is pointed n times, written $(T, V, \mathbf{t})$, where $\mathbf{t}$ is a vector of elements of T of length n. A morphism $(g, h) : (T, V, \mathbf{t}) \to (T', V', \mathbf{t}')$ is a morphism $(g, h) : (T, V) \to (T', V')$ as above, such that $\mathbf{t}' = \mathbf{t} \circ g$.

We call $\mathcal{T}U_n : \mathcal{TC}_n \to \mathsf{Set}$ the forgetful functor associating to any pointed family $(T, V, t_1, \dots, t_n)$ the indexing set T. Note that for a fixed set T, the category $\mathcal{C}_n^T$ (cf. Definition 2.14) is the fibre over T of this functor.

Remark 2.18 (Picking out sorts)**.** Let $1 : \mathcal{TC}_n \to \mathsf{Set}$ denote the constant functor which maps objects to the terminal object of the category Set. A natural transformation $\tau : 1 \Longrightarrow \mathcal{T}U_n$ associates to any object $(T, V, \mathbf{t})$ of the category $\mathcal{TC}_n$ an element of T. Naturality imposes that $\tau(T', V', \mathbf{t}') = g\,(\tau(T, V, \mathbf{t}))$ for any $(g, h) : (T, V, \mathbf{t}) \to (T', V', \mathbf{t}')$.

Notation 2.19. Given a natural transformation $\tau : 1 \Longrightarrow \mathcal{T}U_n$ as in Remark 2.18, we write

$$\tau(T, V, \mathbf{t}) := \tau(T, V, \mathbf{t})(*) \in T \ ,$$

i.e., we omit the argument $* \in 1_{\mathsf{Set}}$ of the singleton set.

Example 2.20. For $1 \leq k \leq n$, we denote by $k : 1 \Longrightarrow \mathcal{T}U_n : \mathcal{TC}_n \to \mathsf{Set}$ the natural transformation such that $k(T, V, \mathbf{t}) := \mathbf{t}(k)$.

We are interested in monads on the category Set^T of families of sets indexed by T and relative monads on $\Delta^T : \mathsf{Set}^T \to \mathsf{Pre}^T$ as well as their relationship:

Definition 2.21 (Relative monads on Δ^T and monads on Set^T)**.** Let P be a relative monad on $\Delta^T : \mathsf{Set}^T \to \mathsf{Pre}^T$. By postcomposing with the forgetful functor $U^T : \mathsf{Pre}^T \to \mathsf{Set}^T$ we obtain a monad

$$UP : \mathsf{Set}^T \to \mathsf{Set}^T \ .$$

The substitution is defined, for $m : X \to UPY$ by setting

$$U\sigma : m \mapsto U\left(\sigma\left(\varphi^{-1} m\right)\right) \ ,$$

making use of the adjunction φ of Remark 2.8. Conversely, to any monad P over Set^T, we associate a relative monad ΔP over Δ^T by postcomposing with Δ^T.

We generalize the definition of colax monad morphisms [Lei04] to relative monads:

Definition 2.22 (Colax morphism of relative monads)**.** Suppose given two relative monads $P : \mathcal{C} \xrightarrow{F} \mathcal{D}$ and $Q : \mathcal{C}' \xrightarrow{F'} \mathcal{D}'$. A *colax morphism of relative monads* from P to Q is a quadruple $h = (G, G', N, \tau)$ of a functor $G \colon \mathcal{C} \to \mathcal{C}'$, a functor $G' : \mathcal{D} \to \mathcal{D}'$ as well as a natural transformation $N : F'G \to G'F$ and a natural transformation $\tau : PG' \Longrightarrow GQ$ such that the following diagrams commute for any objects c, d and any suitable morphism f:

$$\begin{array}{ccc} G'Pc & \xrightarrow{G'\sigma^P(f)} & G'Pd \\ {\scriptstyle \tau_c}\downarrow & & \downarrow{\scriptstyle \tau_d} \\ QGc & \xrightarrow[\sigma^Q(\tau_d \circ G'f \circ Nc)]{} & QGd \end{array} \qquad \begin{array}{ccccc} F'Gc & \xrightarrow{Nc} & G'Fc & \xrightarrow{G'\eta^P_c} & G'Pc \\ & \searrow^{\eta^Q_{Gc}} & & & \downarrow{\scriptstyle \tau_c} \\ & & & & QGc. \end{array}$$

Naturality of the family τ is actually a consequence of the commutative diagrams and may be omitted from the definition.

Remark 2.23. In Section 3 we are going to use the following instance of the preceding definition: the categories $\mathcal{C}$ and $\mathcal{C}'$ are instantiated by Set^T and $\mathsf{Set}^{T'}$, respectively, for sets T and T'. The functor G is the retyping functor (cf. Definition 2.13) associated to some translation of types $g : T \to T'$. Similarly, the categories $\mathcal{D}$ and $\mathcal{D}'$ are instantiated by Pre^T and $\mathsf{Pre}^{T'}$, and the functor F by

$$F := \Delta^T : \mathsf{Set}^T \to \mathsf{Pre}^T \ ,$$

and similar for F':

$$\begin{array}{ccc} \mathsf{Set}^T & \xrightarrow{\Delta^T} & \mathsf{Pre}^T \\ {\scriptstyle \vec{g}}\downarrow & \overset{\mathrm{Id}}{\swarrow\!\!\!\swarrow} & \downarrow{\scriptstyle \vec{g}} \\ \mathsf{Set}^{T'} & \xrightarrow[\Delta^{T'}]{} & \mathsf{Pre}^{T'}. \end{array}$$

Given a monad P on $F : \mathcal{C} \to \mathcal{D}$, the notion of *module over* P generalizes the notion of monadic substitution:

Definition 2.24 (Module over a relative monad)**.** Let $P\colon \mathcal{C} \xrightarrow{F} \mathcal{D}$ be a relative monad and let $\mathcal{E}$ be a category. A *module* M *over* P *with codomain* $\mathcal{E}$ is given by

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

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

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

$$\begin{array}{ccc} Mc & \xrightarrow{\varsigma(f)} & Md \\ & {\scriptstyle \varsigma(\sigma(g)\circ f)}\searrow & \downarrow{\scriptstyle \varsigma(g)} \\ & & Me \end{array} \qquad \begin{array}{ccc} Mc & & \\ & \overset{\mathrm{id}}{\searrow} \quad \overset{\varsigma(\eta_c)}{\nearrow} & \\ & & Mc. \end{array}$$

A functorial action for such a module M is then defined similarly to that for relative monads: for any morphism $f : c \to d$ in $\mathcal{C}$ we set

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

Examples of modules over relative monads are given in the next section.

A *module morphism* is a family of morphisms that is compatible with module substitution in the source and target modules:

Definition 2.25 (Morphism of relative modules)**.** Let M and N be two relative modules over $P\colon \mathcal{C} \xrightarrow{F} \mathcal{D}$ 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 all morphisms $f \in \mathcal{D}(Fc, Pd)$ the following diagram commutes:

$$\begin{array}{ccc} Mc & \xrightarrow{\varsigma^M(f)} & Md \\ {\scriptstyle \rho_c}\downarrow & & \downarrow{\scriptstyle \rho_d} \\ Nc & \xrightarrow[\varsigma^N(f)]{} & Nd. \end{array}$$

The modules over P with codomain $\mathcal{E}$ and morphisms between them form a category called $\mathsf{RMod}(P, \mathcal{E})$. Composition and identity morphisms of modules are defined by pointwise composition and identity.

2.2. **Constructions on modules over relative monads.** The following constructions generalize those used in [HM07b] to modules over *relative* monads.

Any relative monad P comes with the *tautological* module over P itself:

Definition 2.26 (Tautological module)**.** Every relative monad P on $F : \mathcal{C} \to \mathcal{D}$ yields a module (P, σ^P) — also denoted by P — over itself, i.e. an object in the category $\mathsf{RMod}(P, \mathcal{D})$.

Example 2.27 (Example 2.9 continued)**.** The map $\mathsf{TLC}_\beta : V \mapsto \mathsf{TLC}_\beta(V)$ yields a module over the relative monad TLC_β, the *tautological* TLC_β*–module* TLC_β.

Constant functors are modules over relative monads with the same source category:

Definition 2.28 (Constant and terminal module)**.** Let P be a relative monad on $F : \mathcal{C} \to \mathcal{D}$. For any object $e \in \mathcal{E}$ the constant map $T_e \colon \mathcal{C} \to \mathcal{E}$, $c \mapsto e$ for all $c \in \mathcal{C}$, is equipped with the structure of a P–module by setting $\varsigma_{c,d}(f) = \mathrm{id}_e$. In particular, if $\mathcal{E}$ has a terminal object $1_\mathcal{E}$, then the constant module $T_{1_\mathcal{E}} : c \mapsto 1_\mathcal{E}$ is terminal in $\mathsf{RMod}(P, \mathcal{E})$.

Constant modules can be expressed via the following construction:

Definition 2.29 (Postcomposition with a functor)**.** Let P be a relative 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 $M := P$ (considered as tautological module over itself) 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 the preceding definition.

Postcomposition with a functor $F : \mathcal{E} \to \mathcal{X}$ extends to morphisms of modules and yields a functor between categories of modules, $\mathsf{RMod}(P, \mathcal{E}) \to \mathsf{RMod}(P, \mathcal{X})$. We omit the details since we do not use this fact.

The next construction on modules does not change the target category, but the underlying relative monad: given a module N over a relative monad Q and a monad morphism $\tau : P \to Q$ into Q, we rebase or "pull back" the module N along τ, yielding a module over P:

Definition 2.30 (Pullback module)**.** Suppose given two relative monads P and Q and a morphism $h = (G, G', N, \tau) : P \to Q$ as in Definition 2.22. Let M be a Q-module with codomain $\mathcal{E}$. We define a P-module h^*M to $\mathcal{E}$ with object map

$$c \mapsto M(Gc)$$

by defining the substitution map, for $f : Fc \to Pd$, as

$$\varsigma^{h^*M}(f) := \varsigma^M(\tau_d \circ G'f \circ N_c) \ .$$

The module thus defined is called the *pullback module of* M *along* h. The pullback extends to module morphisms and is functorial.

Definition 2.31 (Module morphism induced by a monad morphism)**.** With the same notation as in Definition 2.30, the monad morphism h induces a morphism of P–modules $h : G'P \to h^*Q$. Note that the domain module is the module obtained by postcomposing (the tautological module of) P with G', whereas for (traditional) monads the domain module was just the tautological module of the domain monad [HM07b].

We can take the *product* of two modules:

Definition 2.32 (Products lift)**.** Suppose the category $\mathcal{E}$ has products. Let M and N be P–modules with codomain $\mathcal{E}$. Then the map

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

is canonically equipped with a substitution and thus constitutes a module called the *product of M and N*. This construction extends to a product on $\mathsf{RMod}(P, \mathcal{E})$.

Example 2.33. Given $s, t \in T$, the map $V \mapsto \mathsf{TLC}_\beta(V)(s \rightsquigarrow t) \times \mathsf{TLC}_\beta(V)(s)$ inherits a structure of an TLC_β–module.

2.3. **Derivation & fibre.** We are particularly interested in relative monads on the functor $\Delta^T : \mathsf{Set}^T \to \mathsf{Pre}^T$ for some set T, and modules over such monads. *Derivation* and *fibre*, two important constructions of [HM10] on modules over monads on families of sets, carry over to modules over relative monads on Δ^T.

Definition 2.34. Given $u \in T$, we denote by $D(u) \in \mathsf{Set}^T$ the context with $D(u)(u) = \{*\}$ and $D(u)(t) = \emptyset$ for $u \neq t$. For a context $V \in \mathsf{Set}^T$ we set $V^u := V + D(u)$.

Definition 2.35. Given a monad P over Δ^T and a P–module M with codomain $\mathcal{E}$, we define the derived module of M with respect to $u \in T$ by setting

$$M^u(V) := M(V^u) \ .$$

The module substitution is defined, for $f \in \mathsf{Pre}^T(\Delta^T V, PW)$, by

$$\varsigma^{M^u}(f) := \varsigma^M({}_u f) \ .$$

Here the "shifted" map

$${}_u f \in \mathsf{Pre}^T(\Delta^T(V^u), P(W^u))$$

is the adjunct under the adjunction of Remark 2.8 of the coproduct map

$$\varphi({}_u f) := [P(\mathrm{inl}) \circ f, \eta(\mathrm{inr}(*))] : V^{*u} \to UP(W^u) \ ,$$

where $[\mathrm{inl}, \mathrm{inr}] = \mathrm{id} : W^u \to W^u$. Derivation is an endofunctor on the category of P–modules with codomain $\mathcal{E}$.

Example 2.36. Given $V \in \mathsf{Set}^T$ and $s \in T$, we denote by V^s the context V enriched by an additional variable of type s as in Definition 2.34. The map $\mathsf{TLC}^s_\beta : V \mapsto \mathsf{TLC}_\beta(V^s)$ inherits the structure of a TLC_β–module from the tautological module TLC_β (cf. Example 2.27). We call TLC^s_β the *derived module with respect to $s \in T$* of the module TLC_β; cf. also Section 2.2.

Notation 2.37. In case the set T of types is $T = \{*\}$ the singleton set of types, i.e., when talking about untyped syntax, we denote by M' the derived module of M. Given a natural number n, we denote by M^n the module obtained by deriving n times the module M.

We derive more generally with respect to a natural transformation $\tau : 1 \Longrightarrow \mathcal{T}U_n$ as in Definition 2.17:

Definition 2.38 (Derived module)**.** Let $\tau : 1 \Longrightarrow \mathcal{T}U_n$ be a natural transformation. Let T be a set and P be a relative monad on Δ^T_n. Given any P–module M, we call *derivation of M with respect to τ* the module with object map $M^\tau(V) := M\left(V^{\tau(V)}\right)$.

Definition 2.39. Let P be a relative monad over F, and M a P–module with codomain $\mathcal{E}^T$ for some category $\mathcal{E}$. The *fibre module* $[M]_t$ *of* M *with respect to* $t \in T$ has object map

$$c \mapsto M(c)(t) = M(c)_t$$

and substitution map

$$\varsigma^{[M]_t}(f) := \left(\varsigma^M(f)\right)_t \ .$$

Example 2.40. Given $t \in T$, the map $V \mapsto \mathsf{TLC}_\beta(V)(t) : \mathsf{Set}^T \to \mathsf{Pre}$ inherits a structure of a TLC_β–module, the *fibre module* $[\mathsf{TLC}_\beta]_t$ *with respect to* $t \in T$.

This definition generalizes to fibres with respect to a natural transformation as in Definition 2.38.

Example 2.41 (Examples 2.27, 2.36, 2.33 continued)**.** Abstraction and application are morphisms of TLC_β–modules:

$$\mathrm{Abs}_{s,t} : [\mathsf{TLC}^s_\beta]_t \to [\mathsf{TLC}_\beta]_{s \rightsquigarrow t} \ ,$$
$$\mathrm{App}_{s,t} : [\mathsf{TLC}_\beta]_{s \rightsquigarrow t} \times [\mathsf{TLC}_\beta]_s \to [\mathsf{TLC}_\beta]_t \ .$$

The pullback operation commutes with products, derivations and fibres :

Lemma 2.42. *Let* $\mathcal{C}$ *and* $\mathcal{D}$ *be categories and* $\mathcal{E}$ *be a category with products. Let* $P \colon \mathcal{C} \to \mathcal{D}$ *and* $Q \colon \mathcal{C} \to D$ *be monads over* $F : \mathcal{C} \to \mathcal{D}$ *and* $F' : \mathcal{C}' \to \mathcal{D}'$*, resp., 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 2.43. *Consider the setting as in the preceding lemma, with* $F = \Delta^T$*, and* $t \in T$*. Then we have*

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

This readily generalizes to general derivation as defined in Definition 2.38,

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

Lemma 2.44. *Suppose* N *is a* Q*–module with codomain* $\mathcal{E}^T$*, and* $t \in T$*. Then*

$$\rho^*[M]_t \cong [\rho^* M]_t \ .$$

Definition 2.45 (Substitution of *one* variable)**.** Let T be a (nonempty) set and let P be a *reduction monad* (cf. Definition 2.10) over Δ^T. For any $s, t \in T$ and $X \in \mathsf{Set}^T$ we define a binary substitution operation

$$\mathrm{subst}_{s,t}(X) : P(X^s)_t \times P(X)_s \to P(X)_t,$$
$$(y, z) \mapsto y[* := z] := \sigma\left([\eta_X, \lambda x.z]\right)(y) \ .$$

Here $y : P(X^s)_t$ is of type t and lives in a context X^s, which is X extended by an object variable of object type s. This object variable is substituted with $z : P(X)_s$, which is of type s itself. For any pair $(s, t) \in T^2$, we thus obtain a morphism of P–modules

$$\mathrm{subst}^P_{s,t} : [P^s]_t \times [P]_s \to [P]_t \ .$$

Observe that this substitution operation is monotone in both arguments: monotonicity in the first argument is a consequence of the monadic axioms. Monotonicity in the second argument is a consequence of P being a reduction monads (Definition 2.10).

3. Signatures, models, initiality

We combine the techniques of earlier work [Ahr12a, Ahr16] in order to obtain an initiality result for simple type systems with reductions on the term level. As an example, we specify, via the iteration principle coming from the universal property, a semantically faithful translation from PCF with its usual reduction relation to the untyped lambda calculus with β-reduction.

More precisely, in this section we define a notion of *2–signature* and suitable *models* for such 2–signatures, such that the types and terms generated by the 2–signature, equipped with reduction rules according to the directed equations specified by the 2–signature, form the *initial model*. Analogously to our work on untyped syntax [Ahr16], we define a notion of *2–signature* with two levels: a *syntactic* level specifying types and terms of a language, and, on top of that, a *semantic* level specifying reduction rules on the terms.

3.1. 1–Signatures.

A *1–signature* specifies types and terms over these types. We give two presentations of 1–signatures, a *syntactic* one (cf. Definition 3.7) and a *semantic* one (cf. Definition 3.19). The syntactic presentation is the same as in earlier work [Ahr12a]. The semantic presentation in the present work is adapted from [Ahr12a] to our use of *relative* monads or, to be more precise, *reduction monads*, as compared to traditional monads used in [Ahr12a].

3.1.1. *Signatures for Types.*

We present *type signatures*, which later are used to specify the *object types* of the languages we consider. Such signatures and their models were first considered by Birkhoff [Bir35]. Intuitively, a type signature specifies the respective arities—i.e., the number of arguments—of a set of operations on a set.

Definition 3.1 (Type signature)**.** A *type signature* S is a family of natural numbers, i.e., a set J_S (of operation symbols) and a map (carrying the same name as the signature) $S : J_S \to \mathbb{N}$. For $j \in J_S$ and $n \in \mathbb{N}$, we also write $j : n$ instead of $j \mapsto n$. An element of J resp. its image under S is called an *arity* of S.

Intuitively, the meaning of $j : n$ is that j represents an operation taking n arguments.

Example 3.2 (Type signature of T_{TLC}, Example 2.9)**.** The type signature of the types of the simply–typed lambda calculus is given by

$$S_{\mathsf{TLC}} := \{* : 0 \ , \quad (\rightsquigarrow) : 2\} \ .$$

To any type signature we associate a category of *models*. We call *model of* S any set U equipped with operations according to the signature S. A *morphism of models* is a map between the underlying sets that is compatible with the operations on either side in a suitable sense. Models and their morphisms form a category. We give the formal definitions:

Definition 3.3 (Model of a type signature)**.** A model R of a type signature S is given by

- a set X and
- for each $j \in J_S$, an operation $j^R : X^{S(j)} \to X$.

In the following, given a model R, we write R also for its underlying set.

Definition 3.4 (Morphisms of models)**.** Given two models T and U of the type signature S, a *morphism* from T to U is a map $f : T \to U$ such that, for any arity $j : n$ of S, we have

$$f \circ j^T = j^U \circ (\underbrace{f \times \ldots \times f}_{n \text{ times}}) \ .$$

Example 3.5. The language PCF [Plo77, HO00] is a simply–typed lambda calculus with a fixed point operator and arithmetic constants. Let $J := \{\iota, o, (\Rightarrow)\}$. The signature of the types of PCF is given by the arities

$$S_{\mathsf{PCF}} := \{\iota : 0 \ , \quad o : 0 \ , \quad (\Rightarrow) : 2\} \ .$$

A model T of S_{PCF} is given by a set T and three operations,

$$\iota^T : T \ , \quad o^T : T \ , \quad (\Rightarrow)^T : T \times T \to T \ .$$

Given two models T and U of S_{PCF}, a morphism from T to U is a map $f : T \to U$ between the underlying sets such that, for any $s, t \in T$,

$$\begin{aligned} f(\iota^T) &= \iota^U \ , \\ f(o^T) &= o^U \quad \text{and} \\ f(s \Rightarrow^T t) &= f(s) \Rightarrow^U f(t) \ . \end{aligned}$$

3.1.2. *Signatures for terms.* Before giving our definition of signature for terms, we consider the example of the simply-typed lambda calculus and describe our goals using this example.

Let T_{TLC} be the initial model of the signature for types of Example 3.2. The signature for simply-typed lambda terms over those types may be given as follows:

$$\{\mathrm{abs}_{s,t} := [([s], t)] \to (s \rightsquigarrow t) \ , \quad \mathrm{app}_{s,t} := [([], s \rightsquigarrow t), ([], s)] \to t\}_{s,t \in T_{\mathsf{TLC}}} \ . \tag{3.1}$$

Intuitively, this means that we have two families of operations, abs—abstraction—and app—application. The abstraction (specified by) $\mathrm{abs}_{s,t}$ takes one argument, specified by a list of length one, where this argument is of type t and lives in a context extended by a variable of type s. The return type is $s \rightsquigarrow t$.

The parameters s and t range over the set T_{TLC} of types, the initial model of the signature for types from Example 3.2. Our goal is to consider models of the simply–typed lambda calculus in monads over categories of the form Set^T for *any* set T—provided that T is equipped with a model of the signature S_{TLC}. It thus is more suitable to specify the signature of the simply–typed lambda calculus as follows:

$$\{\mathrm{abs} := [([1], 2)] \to (1 \rightsquigarrow 2) \ , \quad \mathrm{app} := [([], 1 \rightsquigarrow 2), ([], 1)] \to 2\} \ . \tag{3.2}$$

For any model T of S_{TLC}, the variables 1 and 2 range over elements of T. In this way the number of abstractions and applications depends on the model T of S_{TLC}: intuitively, a model of the above signature of Display (3.2) over a model T of T_{TLC} has T^2 abstractions and T^2 applications—one for each pair of elements of T.

To abstractly capture this kind of arities given in terms of natural numbers, we introduce a notion of *degree*—given by a natural number $n \in \mathbb{N}$—, and term arities of degree n. A result stated towards the end of this section, Lemma 3.16, explains how the "grouping" and "ungrouping" of such arities works.

We start by giving a very syntactic notion of arity for terms, in Definition 3.7. This notion is later related to a more semantic notion, given in Definition 3.18.

Definition 3.6 (Type of degree n)**.** For $n \geq 1$, we call *types of S of degree n* the elements of the set $S(n)$ of types associated to the signature S with free variables in the set $\{1, \dots, n\}$. We set $S(0) := \hat{S}$. Formally, the set $S(n)$ may be obtained as the initial model of the type signature S enriched by n nullary arities.

Types of degree n are used to form elementary arities of degree n:

Definition 3.7 (Elementary arity of degree n)**.** An elementary arity for terms over the signature S for types of degree n is of the form

$$[([t_{1,1}, \dots, t_{1,m_1}], t_1), \dots, ([t_{k,1}, \dots, t_{k,m_k}], t_k)] \to t_0 \ , \tag{3.3}$$

where $t_{i,j}, t_i \in S(n)$. More formally, an elementary arity of degree n over S is a pair consisting of an element $t_0 \in S(n)$ and a list of pairs. where each pair itself consists of a list $[t_{i,1}, \dots, t_{i,m_i}]$ of elements of $S(n)$ and an element t_i of $S(n)$.

An elementary arity of the form given in Display (3.3) denotes a constructor—or a family of constructors, for $n > 0$—whose output type is t_0, and whose k inputs are terms of type t_i, respectively. In the input i, for $1 \leq i \leq k$, variables of type according to the list $[t_{i,1}, \dots, t_{i,m_i}]$ are bound by the constructor.

Example 3.8. The prototypical examples of elementary arities of degree 2 are abstraction and application of Display (3.2).

Compared to our work on pure syntax [Ahr12a] we have to adapt the *semantic* definition of signatures for terms, since we now work with reduction monads on Δ^T for some set T instead of monads over families of sets.

Definition 3.9 (Relative S–Monad)**.** Given a type signature S, the *category S-RMnd of relative S–monads* is defined as the category whose objects are pairs (T, P) of a model T of S and a reduction monad

$$P : \mathsf{Set}^T \xrightarrow{\Delta^T} \mathsf{Pre}^T \ .$$

A morphism from (T, P) to (T', P') is a pair (g, f) of a morphism of S–models $g : T \to T'$ and a morphism of relative monads $f : P \to P'$ over the retyping functor $\vec{g}$ as in Remark 2.23.

Given $n \in \mathbb{N}$, we write S-RMnd_n for the category whose objects are pairs (T, P) of a model T of S and a reduction monad P over Δ_n^T. A morphism from (T, P) to (T', P') is a pair (g, f) of a morphism of S–model $g : T \to T'$ and a monad morphism $f : P \to P'$ over the retyping functor $\vec{g}_n$ defined in Definition 2.16.

In the following we need a category in which we gather modules over *different* relative monads. More precisely, an object is given by a pair of a relative monad and a module on it. A morphism between two such pairs is given by a pair of a monad morphism and a module morphism, where for the latter we use the pullback operation on modules to obtain modules over the same relative monad:

Definition 3.10 (Large category $\mathsf{LRMod}_n(S, \mathcal{D})$ of modules)**.** Given a natural number $n \in \mathbb{N}$, a type signature S and a category $\mathcal{D}$, we call $\mathsf{LRMod}_n(S, \mathcal{D})$ the category an object of which is a pair (P, M) of a relative S–monad $P \in S$-RMnd_n and a P–module with codomain $\mathcal{D}$. A morphism to another such (Q, N) is a pair (f, h) of a morphism of relative S–monads $f : P \to Q$ in S-RMnd_n and a morphism of relative modules $h : M \to f^* N$.

We sometimes just write the module—i.e. the second—component of an object or morphism of the large category of modules. Given $M \in \mathsf{LRMod}_n(S, \mathcal{D})$, we thus write $M(V)$ or M_V for the value of the module on the object V.

Let S be a type signature. A *half-arity over S of degree n* is a functor from relative S-monads to the category of large modules of degree n:

Definition 3.11 (Half-Arity over S (of degree n))**.** Given a type signature S and $n \in \mathbb{N}$, we call *half-arity over S of degree n* a functor

$$\alpha : S\text{-}\mathsf{RMnd} \to \mathsf{LRMod}_n(S, \mathsf{Pre}) \ .$$

which is a section to the forgetful functor forgetting both the module and the "points".

Intuitively, a half-arity of degree n associates, to any S-monad P, a module over P_n.

We restrict ourselves to a class of such functors, starting with the *tautological* module:

Definition 3.12 (Tautological module of degree n)**.** Given $n \in \mathbb{N}$, any relative monad R over Δ^T induces a monad R_n over Δ_n^T with object map $(V, t_1, \ldots, t_n) \mapsto (RV, t_1, \ldots, t_n)$. To any relative S–monad R we associate the tautological module of R_n,

$$\Theta_n(R) := (R_n, R_n) \in \mathsf{LRMod}_n(S, \mathsf{Pre}_n^T) \ .$$

Furthermore, we use *canonical natural transformations* (cf. Definition 3.14) to build *elementary* half–arities; these transformations specify context extension (derivation) and selection of specific object types (fibre):

Definition 3.13 ($S\mathcal{C}_n$)**.** Given a type signature S, a natural number $n \in \mathbb{N}$, and a category $\mathcal{C}$, we define the category $S\mathcal{C}_n$ to be the category with, as objects, diagrams of the form

$$n \xrightarrow{\mathbf{t}} T \xrightarrow{V} \mathcal{C} \ ,$$

written $(V, t_1, \ldots, t_n)$ with $t_i := \mathbf{t}(i)$, where T is a model of S, the object $V \in \mathcal{C}^T$ is a T–indexed family of objects of $\mathcal{C}$ and $\mathbf{t}$ is a vector of elements of T of length n. A morphism h to another such $(W, \mathbf{t})$ with the same pointing map $\mathbf{t}$ is given by a morphism $h : V \to W$ in $\mathcal{C}^T$ whose first component is a morphism of S-models; there are no morphisms between objects with different pointing maps.

We denote by $SU_n : S\mathcal{C}_n \to \mathsf{Set}$ the functor mapping an object $(T, V, \mathbf{t})$ to the underlying set T.

We have a forgetful functor $S\mathcal{C}_n \to \mathcal{TC}_n$ which forgets the structure of model. On the other hand, any model T of S in a set T gives rise to a functor $\mathcal{C}_n^T \to S\mathcal{C}_n$, which "attaches" the structure of model.

The meaning of a term $s \in S(n)$ as a natural transformation

$$s : 1 \Longrightarrow SU_n : S\mathcal{C}_n \to \mathsf{Set}$$

is now given by recursion on the structure of s:

Definition 3.14 (Canonical natural transformation)**.** Let $s \in S(n)$ be a type of degree n. Then s denotes a natural transformation

$$s : 1 \Longrightarrow SU_n : S\mathcal{C}_n \to \mathsf{Set}$$

defined recursively on the structure of s as follows: for $s = \alpha(a_1, \ldots, a_k)$ the image of a constructor $\alpha \in S$ we set

$$s(T, V, \mathbf{t}) = \alpha(a_1(T, V, \mathbf{t}), \ldots, a_k(T, V, \mathbf{t}))$$

and for $s = m$ with $1 \leq m \leq n$ we define

$$s(T, V, \mathbf{t}) = \mathbf{t}(m) \ .$$

We call a natural transformation of the form $s \in S(n)$ *canonical*.

Definition 3.15 (Elementary half-arity)**.** We restrict our attention to *elementary* half–arities, which we define analogously to [Ahr12a] as constructed using derivations and products, starting from the fibres of the tautological module and the constant singleton module. The following clauses define the inductive set of elementary half-arities:

- The constant functor $* : R \mapsto 1$ is an elementary half–arity.
- Given any canonical natural transformation $\tau : 1 \Longrightarrow SU_n$ (cf. Definition 3.14), the point-wise fibre module with respect to τ (cf. Definition 2.39) of the tautological module $\Theta_n : R \mapsto (R_n, R_n)$ (cf. Definition 3.12) is an elementary half–arity of degree n,
$$[\Theta_n]_\tau : S\text{-}\mathsf{RMnd} \to \mathsf{LRMod}_n(S, \mathsf{Set}) \ , \quad R \mapsto [R_n]_\tau \ .$$
- Given any elementary half-arity $M : S\text{-}\mathsf{Mnd} \to \mathsf{LMod}_n(S, \mathsf{Set})$ of degree n and a canonical natural transformation $\tau : 1 \Longrightarrow SU_n$, the point-wise derivation of M with respect to τ is an elementary half–arity of degree n,
$$M^\tau : S\text{-}\mathsf{RMnd} \to \mathsf{LRMod}_n(S, \mathsf{Set}) \ , \quad R \mapsto \big(M(R)\big)^\tau \ .$$
Here $\big(M(R)\big)^\tau$ really means derivation of the module, i.e., derivation in the second component of $M(R)$.
- For a half–arity M, let $M_i : R \mapsto \pi_i M(R)$ denote the ith projection. Given two elementary half–arities M and N of degree n, which coincide pointwise on the first component, i.e. such that $M_1 = N_1$. Then their product $M \times N$ is again an elementary half-arity of degree n. Here the product is really the pointwise product in the second component, i.e.
$$M \times N : R \mapsto \big(M_1(R), M_2(R) \times N_2(R)\big) \ .$$

As explained at the beginning of Section 3.1.2, our goal is to consider arities such as application, which are families of arities indexed by object types, as one arity of higher degree. The following result explains how this grouping (and ungrouping) translates to half–arities: a half–arity of degree n thus associates, to any relative S–monad P over a set of types T, a *family of P–modules* indexed by T^n.

Lemma 3.16 (Module of higher degree corresponds to a family of modules)**.** *Let $\mathcal{C}$ be a category, let T be a set and R be a monad on $\mathcal{C}^T$. Suppose $n \in \mathbb{N}$, and let $\mathcal{D}$ be a category. Then modules over R_n with codomain $\mathcal{D}$ correspond precisely to families of R–modules indexed by T^n with codomain $\mathcal{D}$ by (un)currying. More precisely, let M be an R_n–module. Given $\mathbf{t} \in T^n$, we define an R–module $M_{\mathbf{t}}$ by*

$$M_{\mathbf{t}}(c) := M(c, \mathbf{t}) \ .$$

Module substitution for $M_{\mathbf{t}}$ is given, for $f \in \mathcal{C}^T(c, Rd)$, by

$$\varsigma^{M_{\mathbf{t}}}(f) := \varsigma^M(f)$$

where we use that we also have $f \in \mathcal{C}_n^T((c, \mathbf{t}), (Rd, \mathbf{t}))$ according to Definition 2.14. Going the other way round, given a family $(M_{\mathbf{t}})_{\mathbf{t} \in T^n}$, we define the R_n–module M by

$$M(c, \mathbf{t}) := M_{\mathbf{t}}(c) \ .$$

Given a morphism $f \in \mathcal{C}_n^T((c,\mathbf{t}),(Rd,\mathbf{t}))$ *— recall that morphisms in* $\mathcal{C}_n^T$ *are only between families with* the same marker $\mathbf{t}$ *—, we also have* $f \in \mathcal{C}^T(c,Rd)$ *and define*

$$\varsigma^M(f) := \varsigma^{M_{\mathbf{t}}}(f) \ .$$

The remark extends to morphisms of modules; indeed, a morphism of modules $\alpha : M \to N$ *on categories with pointed index sets corresponds to a family of morphisms* $(\alpha_{\mathbf{t}} : M_{\mathbf{t}} \to N_{\mathbf{t}})_{\mathbf{t}\in T^n}$ *between the associated families of modules.*

An arity of degree $n \in \mathbb{N}$ for terms over a type signature S is defined to be a pair of functors from relative S–monads to modules in $\mathsf{LRMod}_n(S,\mathsf{Pre})$. The degree n corresponds to the number of object type indices of its associated constructor. As an example, the arities of abs and app of Display (3.2) are of degree 2.

Definition 3.17 (Weighted set)**.** A weighted set is a set J together with a map $d : J \to \mathbb{N}$.

Definition 3.18 (Term-arity, signature over S)**.** An *elementary arity* α *over* S *of degree* n is a pair

$$s = (\mathrm{dom}(\alpha), \mathrm{cod}(\alpha))$$

of half–arities over S of degree n such that

- $\mathrm{dom}(\alpha)$ is elementary and
- $\mathrm{cod}(\alpha)$ is of the form $[\Theta_n]_\tau$ for some canonical natural transformation τ as in Definition 3.14.

Any elementary arity is thus *syntactically* of the form given in Definition 3.7. We write $\mathrm{dom}(\alpha) \to \mathrm{cod}(\alpha)$ for the arity α, and $\mathrm{dom}(\alpha,R) := \mathrm{dom}(\alpha)(R)$ and similar for the codomain and morphisms of relative S–monads. Given a weighted set (J,d) as in Definition 3.17, a *term–signature* Σ *over* S *indexed by* (J,d) is a J-family Σ of elementary arities over S, the arity $\Sigma(j)$ being of degree $d(j)$ for any $j \in J$.

Definition 3.19 (1–signature)**.** A *1–signature* is a pair (S,Σ) consisting of a type signature S and a term–signature Σ (indexed by some weighted set) over S. By definition, the term-arities of Σ are elementary.

Example 3.20. The terms of the simply typed lambda calculus over the type signature of Example 3.2 are given by the arities

$$\mathrm{abs} : [(\Theta_2)^1]_2 \to [\Theta_2]_{1\rightsquigarrow 2} \ ,$$
$$\mathrm{app} : [\Theta_2]_{1\rightsquigarrow 2} \times [\Theta_2]_1 \to [\Theta_2]_2 \quad ,$$

both of which are of degree 2. Contrast that to the signature for the simply–typed lambda calculus we gave in Display (3.1). The difference is that here all the family of "application" arities, indexed by pairs of object types, are grouped together into one arity of degree 2—and similar for abstraction—whereas this is not the case in Display (3.1).

Those two arities, abs and app, can in fact be considered over any type signature S with an arrow constructor, in particular over the signature S_{PCF} (cf. Example 3.21).

Example 3.21 (Example 3.5 continued)**.** We continue considering PCF. The signature S_{PCF} for its types is given in Example 3.5. The term–signature of PCF is given in Figure 1. It consists of an arity for abstraction and an arity for application, each of degree 2, an arity (of degree 1) for the fixed point operator, and one arity of degree 0 for each logic and arithmetic constant, some of which we omit.

$$
\begin{aligned}
\mathrm{abs} &: [(\Theta_2)^1]_2 \to [\Theta_2]_{1\Rightarrow 2} \ , \\
\mathrm{app} &: [\Theta_2]_{1\Rightarrow 2} \times [\Theta_2]_1 \to [\Theta_2]_2 \ , \\
\mathbf{Fix} &: [\Theta_1]_{1\Rightarrow 1} \to [\Theta_1]_1 \ , \\
\mathbf{n} &: * \to [\Theta]_\iota \qquad \text{for } n \in \mathbb{N} \\
\mathbf{Succ} &: * \to [\Theta]_{\iota\Rightarrow\iota} \\
\mathbf{Pred} &: * \to [\Theta]_{\iota\Rightarrow\iota} \\
\mathbf{Zero?} &: * \to [\Theta]_{\iota\Rightarrow o} \\
\mathbf{cond}_\iota &: * \to [\Theta]_{o\Rightarrow\iota\Rightarrow\iota\Rightarrow\iota} \\
\mathbf{T}, \mathbf{F} &: * \to [\Theta]_o \\
&\vdots
\end{aligned}
$$

Figure 1: Term signature of PCF

3.2. **Models of 1–signatures.** Before giving the general definition, we explain what a model of the simply–typed lambda calculus looks like according to our definitions:

Example 3.22. A model of $(S_{\mathsf{TLC}}, \Sigma_{\mathsf{TLC}})$ is given by

- a model of S_{TLC} in a set T;
- a reduction monad R on Δ^T;
- for any pair $(s,t) \in T^2$, two morphisms of modules

$$R_t^s \to R_{s\rightsquigarrow t} \quad \text{and} \quad R_{s\rightsquigarrow t} \times R_s \to R_t \ .$$

Note that in the above example, we have "ungrouped" the operations as explained at the beginning of Section 3 and Lemma 3.16.

Most of the work for giving a general definition of model of an arity is already done; the specification is contained in the definition of signature.

Definition 3.23 (Model of an arity, of a signature over S)**.** A *model of an arity α over S in an S–monad R* is a morphism of relative modules

$$\mathrm{dom}(\alpha, R) \to \mathrm{cod}(\alpha, R) \ .$$

A *model R of a signature Σ over S* is a given by a relative S–monad—called R as well—and a model α^R of each arity α of Σ in R.

Models of (S, Σ) are the objects of a category $\mathrm{Model}^\Delta(S, \Sigma)$, whose morphisms are defined as follows:

Definition 3.24 (Morphism of models)**.** Given models P and R of a typed signature (S, Σ), a morphism of models $f : P \to R$ is given by a morphism of relative S–monads $f : P \to R$, such that for any arity α of S the following diagram of module morphisms commutes:

$$
\begin{array}{ccc}
\mathrm{dom}(\alpha, P) & \xrightarrow{\alpha^P} & \mathrm{cod}(\alpha, P) \\
{\scriptstyle \mathrm{dom}(\alpha, f)}\downarrow & & \downarrow{\scriptstyle \mathrm{cod}(\alpha, f)} \\
\mathrm{dom}(\alpha, R) & \xrightarrow[\alpha^R]{} & \mathrm{cod}(\alpha, R).
\end{array}
$$

Lemma 3.25. *For any typed signature (S, Σ), the category of models of (S, Σ) has an initial object.*

Proof. The initial object is obtained, analogously to the untyped case (cf. [Ahr16]), via an adjunction $\Delta_* \dashv U_*$ between the categories of models of (S, Σ) in relative monads and those in monads as in [Ahr12a].

In more detail, to any relative S–monad $(T, P) \in S\text{-}\mathsf{RMnd}$ we associate the S–monad $U(T, P) := (T, UP)$ where U_*P is the monad obtained by postcomposing with the forgetful functor $U^T : \mathsf{Pre}^T \to \mathsf{Set}^T$. Substitution for U_*P is defined, in each fibre, as in Definition 2.21. For any arity $s \in \Sigma$ we have that

$$U_* \operatorname{dom}(s, P) \cong \operatorname{dom}(s, U_*P) \ ,$$

and similar for the codomain. The postcomposed model morphism $U_*s(P)$ hence models s in U_*P in the sense of [Ahr12a]. This defines the functor $U_* : \mathrm{Model}^{\Delta}(S, \Sigma) \to \mathrm{Model}(S, \Sigma)$. Conversely, to any S–monad we can associate a relative S–monad by postcomposing with $\Delta^T : \mathsf{Set}^T \to \mathsf{Pre}^T$, analogous to the untyped case in [Ahr16], yielding $\Delta_* : \mathrm{Model}(S, \Sigma) \to \mathrm{Model}^{\Delta}(S, \Sigma)$. In summary, the natural isomorphism

$$\varphi_{R,P} : (\mathrm{Model}^{\Delta}(S, \Sigma))(\Delta_* R, P) \cong (\mathrm{Model}(S, \Sigma))(R, U_*P)$$

is given by postcomposition with the forgetful functor (from left to right) resp. with the functor Δ (from right to left).

□

Remark 3.26 (on intrinsic vs. extrinsic typing)**.** Our approach to signatures and their models uses *intrinsic typing* [BHKM12]. This term expresses a method of defining exactly the well–typed terms by organizing them into a family of sets parametrized by object types. It is to be contrasted to *ex*trinsic typing, where one starts out with potentially ill-typed "preterms", which then need to be filtered via a well-typedness predicate. Intrinsic typing has two advantages over extrinsic typing:

- Intrinsically typed syntax comes with a more useful recursion principle, which allows for the specification of translations between languages that are *automatically* compatible with typing.
- Intrinsic typing delegates object level typing to the meta language type system, such as the Coq type system. In this way, the meta level type checker (e.g. Coq) sorts out ill–typed terms automatically: writing such a term yields a type error on the meta level.

3.3. **Directed equations.** Analogously to the untyped case (cf. [Ahr16]), a directed equation associates, to any model of (S, Σ) in a relative monad P, two parallel morphisms of P–modules. However, as for arities, a directed equation may be, more precisely, a *family of directed equations*, indexed by object types. As an example, consider the simply–typed lambda calculus from above, which is defined with *typed* abstraction and application. Similarly, we have a *typed substitution* operation for TLC, which substitutes a term of type $s \in T_{\mathsf{TLC}}$ for a free variable of type s in a term of type $t \in T_{\mathsf{TLC}}$, yielding again a term of type t. For $s, t \in T_{\mathsf{TLC}}$ and $M \in \mathsf{TLC}(V + s)_t$ and $N \in \mathsf{TLC}(V)_s$, β-reduction is specified by

$$(\lambda_{s,t} M)N \rightsquigarrow M[* := N] \ ,$$

where our notation hides the fact that not only abstraction, but also application and substitution are typed operations. More formally, such a reduction rule might read as a family of inequalities between morphisms of modules

$$\mathrm{app}_{s,t} \circ (\mathrm{abs}_{s,t} \times \mathrm{id}) \;\; \leq \;\; \_[*^s :=_t \_] \;\; ,$$

where $s, t \in T_{\mathsf{TLC}}$ range over types of the simply–typed lambda calculus. Analogously to Section 3.1.2, we want to specify the β-rule without referring to the set T_{TLC}, but instead express it for an arbitrary model R of the typed signature $(S_{\mathsf{TLC}}, \Sigma_{\mathsf{TLC}})$ (cf. Examples 3.2, 3.20), as in

$$\mathrm{app}^R \circ (\mathrm{abs}^R \times \mathrm{id}) \;\; \leq \;\; \_[* := \_] \;\; ,$$

where both the left and the right side of the inequality are given by suitable R–module morphisms of degree 2. Source and target of a *half–equation* accordingly are given by functors from models of a typed signature (S, Σ) to a suitable category of modules. A half–equation then is a natural transformation between functors into categories of modules:

Definition 3.27 (Category of half-equations)**.** Let (S, Σ) be a signature. An (S, Σ)*–module* U of degree $n \in \mathbb{N}$ is a functor from the category of models of (S, Σ) as defined in Section 3.2 to the category $\mathsf{LRMod}_n(S, \mathsf{Pre})$ (cf. Definition 3.10) commuting with the forgetful functor to the category of relative monads. We define a morphism of (S, Σ)–modules to be a natural transformation which becomes the identity when composed with the forgetful functor. We call these morphisms *half–equations* (of degree n). We write $U^R := U(R)$ for the image of the model R under the S–module U, and similar for morphisms.

Definition 3.28 (Substitution as half-equation)**.** Given a relative monad on Δ^T, its associated substitution–of–one–variable operation (cf. Definition 2.45) yields a family of module morphisms, indexed by pairs $(s, t) \in T$. By Lemma 3.16 this family is equivalent to a module morphism of degree 2. The assignment

$$\mathrm{subst} : R \mapsto \mathrm{subst}^R : [R_2^1]_2 \times [R_2]_1 \to [R_2]_2$$

thus yields a half–equation of degree 2 over any signature S. Its domain and codomain are elementary.

Example 3.29 (Example 3.20 continued)**.** The map

$$\mathrm{app} \circ (\mathrm{abs} \times \mathrm{id}) : R \mapsto \mathrm{app}^R \circ (\mathrm{abs}^R, \mathrm{id}^R) : [R_2^1]_2 \times [R_2]_1 \to [R_2]_2$$

is a half–equation over the signature of TLC, as well as over the signature of PCF.

Definition 3.30. Any elementary arity of degree n,

$$s = [\Theta_n^{\tau_1}]_{\sigma_1} \times \ldots \times [\Theta_n^{\tau_m}]_{\sigma_m} \to [\Theta_n]_\sigma \;\; ,$$

defines an elementary S–module

$$\mathrm{dom}(s) : R \mapsto [R_n^{\tau_1}]_{\sigma_1} \times \ldots \times [R_n^{\tau_m}]_{\sigma_m} \;\; .$$

Definition 3.31 (Directed equation, rule)**.** Given a signature (S, Σ), a *directed equation over* (S, Σ), or (S, Σ)*-rule*, of degree $n \in \mathbb{N}$ is a pair of parallel half–equations between (S, Σ)–modules of degree n. We write $\alpha \leq \gamma$ for the rule (α, γ).

$$\begin{aligned}
\mathbf{Fix} &\leq \mathrm{app} \circ (\mathrm{id}, \mathbf{Fix}) : [\Theta_1]_{1\Rightarrow 1} \to [\Theta_1]_1 \\
\mathrm{app} \circ (\mathbf{Succ}, \mathbf{n}) &\leq \mathbf{n} + \mathbf{1} : * \to [\Theta]_\iota \\
\mathrm{app} \circ (\mathbf{Pred}, \mathbf{0}) &\leq \mathbf{0} : * \to [\Theta]_\iota \\
\mathrm{app} \circ (\mathbf{Pred}, \mathrm{app} \circ (\mathbf{Succ}, \mathbf{n})) &\leq \mathbf{n} : * \to [\Theta]_\iota \\
\mathrm{app} \circ (\mathbf{Zero?}, \mathbf{0}) &\leq \mathbf{T} : * \to [\Theta]_o \\
\mathrm{app} \circ (\mathbf{Zero?}, \mathrm{app} \circ (\mathbf{Succ}, \mathbf{n})) &\leq \mathbf{F} : * \to [\Theta]_o \\
&\vdots
\end{aligned}$$

Figure 2: Reduction Rules of PCF

Example 3.32 (β-reduction)**.** For any 1–signature that has an arity for abstraction and an arity for application, we specify β-reduction using the parallel half–equations of Definition 3.28 and Example 3.29:

$$\mathrm{app} \circ (\mathrm{abs} \times \mathrm{id}) \leq \mathrm{subst} : [\Theta_2^1]_2 \times [\Theta_2]_1 \to [\Theta_2]_2 \ .$$

Example 3.33 (Fixpoints and arithmetics of PCF)**.** The reduction rules of PCF are specified by the rules—over the 1-signature of PCF as given in Example 3.21—of Figure 2.

Given a set A of (S, Σ)-rules, a model of those rules is any model of (S, Σ) which satisfies the rules of A in the following sense:

Definition 3.34 (Models of rules)**.** A *model of an* (S, Σ)*-rule* $\alpha \leq \gamma : U \to V$ (of degree n) is any model R over a set of types T of (S, Σ) such that $\alpha^R \leq \gamma^R$ pointwise, i.e., if for any pointed context $(X, \mathbf{t}) \in \mathsf{Set}^T \times T^n$, any $t \in T$ and any $y \in U^R_{(X,\mathbf{t})}(t)$,

$$\alpha^R(y) \ \leq \ \gamma^R(y) \ , \tag{3.4}$$

where we omit the sort argument t as well as the context $(X, \mathbf{t})$ from α and γ. We say that such a model R *satisfies* the rule $\alpha \leq \gamma$.

For a set A of (S, Σ)-rules, we call *model of* $((S, \Sigma), A)$ any model of (S, Σ) that satisfies each rule of A. We define the category of models of the 2-signature $((S, \Sigma), A)$ to be the full subcategory of the category of models of S whose objects are models of $((S, \Sigma), A)$.

According to Lemma 3.16, the inequality of Display (3.4) is equivalent to ask whether, for any $\mathbf{t} \in T^n$, any $t \in T$ and any $y \in U^R_{\mathbf{t}}(X)(t)$,

$$\alpha^R_{\mathbf{t}}(y) \ \leq \ \gamma^R_{\mathbf{t}}(y) \ .$$

3.4. **Initiality for 2–Signatures.** We are ready to state and prove an initiality result for typed signatures with directed equations:

Theorem 3.35. *For any set of elementary* (S, Σ)*-rules* A*, the category of models of* $((S, \Sigma), A)$ *has an initial object.*

Proof. The basic ingredients for building the initial model are given by the initial model $(\hat{S}, \hat{\Sigma})$—or just $\hat{\Sigma}$ for short—in the category $\mathrm{Model}(S, \Sigma)$ of models in monads on set families [Ahr12a]. Equivalently, the ingredients come from the initial object $(\hat{S}, \Delta_* \hat{\Sigma})$—or just $\Delta_* \hat{\Sigma}$

for short—of models *without rules* in the category $\mathrm{Model}^{\Delta}(S, \Sigma)$ (cf. Lemma 3.25). We call $\hat{\Sigma}$ resp. $\Delta_* \hat{\Sigma}$ the monad resp. relative monad underlying the initial model.

The proof consists of several steps: at first, we define a preorder $\leq_A$ on the terms of $\hat{\Sigma}$, induced by the set A of rules. Afterwards we show that the data of the model $\hat{\Sigma}$—substitution, model morphisms etc.—is compatible with the preorder $\leq_A$ in a suitable sense. This yields a model $\hat{\Sigma}_A$ of (Σ, A). Finally we show that $\hat{\Sigma}_A$ is the initial such model.

The monad underlying the initial model: For any context $X \in \mathsf{Set}^{\hat{S}}$ and $t \in \hat{S}$, we equip $\hat{\Sigma}X(t)$ with a preorder A by setting (*morally*, cf. below), for $x, y \in \hat{\Sigma}X(t)$,

$$x \leq_A y \quad :\Leftrightarrow \quad \forall R : \mathrm{Model}(\Sigma, A), \quad i_R(x) \leq_R i_R(y) \ , \tag{3.5}$$

where $i_R : \Delta_* \hat{\Sigma} \to R$ is the initial morphism in the category of models of (S, Σ), cf. Lemma 3.25. Note that the above definition in Display (3.5) is ill–typed: we have $x \in \hat{\Sigma}X(t)$, which cannot be applied to (a fibre of) $i_R(X) : \vec{g}(\hat{\Sigma}X) \to R(\vec{g}X)$. We denote by $\varphi = \varphi_R$ the natural isomorphism induced by the adjunction of Definition 2.13 obtained by retyping along the initial morphism of types $g : \hat{S} \to T = T_R$ towards the set T of "types" of R,

$$\varphi_{X,Y} : \mathsf{Pre}^{T}\left(\vec{g}(\hat{\Sigma}X), R(\vec{g}X)\right) \cong \mathsf{Pre}^{\hat{S}}\left(\hat{\Sigma}X, R(\vec{g}X) \circ g\right) \ .$$

Instead of the above definition in Display (3.5), we should really write

$$x \leq_A y \quad :\Leftrightarrow \quad \forall R : \mathrm{Model}(\Sigma, A), \quad (\varphi(i_{R,X}))\,(x) \leq_R (\varphi(i_{R,X}))\,(y) \ , \tag{3.6}$$

where we omit the subscript "R" from φ. We have to show that the map

$$X \mapsto \hat{\Sigma}_A X := (\hat{\Sigma}X, \leq_A)$$

yields a relative monad on $\Delta^{\hat{S}}$. The missing fact to prove is that the substitution with a morphism

$$f \in \mathsf{Pre}^{\hat{S}}(\Delta X, \hat{\Sigma}_A Y) \cong \mathsf{Set}^{\hat{S}}(X, \hat{\Sigma}Y)$$

is compatible with the order $\leq_A$: given any $f \in \mathsf{Pre}^{\hat{S}}(\Delta X, \hat{\Sigma}_A Y)$ we show that

$$\sigma^{\hat{\Sigma}}(f) \in \mathsf{Set}^{\hat{S}}(\hat{\Sigma}X, \hat{\Sigma}Y)$$

is monotone with respect to $\leq_A$ and hence (the carrier of) a morphism

$$\sigma^{\hat{\Sigma}_A}(f) \in \mathsf{Pre}^{\hat{S}}(\hat{\Sigma}_A X, \hat{\Sigma}_A Y) \ .$$

We overload the infix symbol $\gg\!=$ to denote monadic substitution. Note that this notation now hides an implicit argument giving the sort of the term in which we substitute. Suppose $x, y \in \hat{\Sigma}X(t)$ with $x \leq_A y$, we show

$$x \gg\!= f \ \ \leq_A \ \ y \gg\!= f \ .$$

Using the definition of $\leq_A$, we must show, for a given model R of (Σ, A),

$$(\varphi(i_R))\,(x \gg\!= f) \ \ \leq_R \ \ (\varphi(i_R))\,(y \gg\!= f) \ . \tag{3.7}$$

Let g be the initial morphism of types towards the types of R. Since $i := i_R$ is a morphism of models and thus in particular a *monad* morphism, it is compatible with the substitution of $\hat{\Sigma}$ and R; we have

$$\begin{array}{ccc} \vec{g}(\hat{\Sigma}X) & \xrightarrow{\vec{g}(\sigma(f))} & \vec{g}(\hat{\Sigma}Y) \\ \Big\downarrow{\scriptstyle i_X} & & \Big\downarrow{\scriptstyle i_Y} \\ R(\vec{g}X) & \xrightarrow[\sigma^R(i_Y \circ \vec{g}f)]{} & R(\vec{g}Y). \end{array} \tag{3.8}$$

By applying the isomorphism φ on the diagram of Display (3.8), we obtain

$$\begin{aligned} \varphi(i_Y) \circ \sigma(f) &= \varphi\left(i_Y \circ \vec{g}(\sigma(f))\right) \\ &= \varphi\left(\sigma(i_Y \circ \vec{g}f) \circ i_X\right) \\ &= g^*\left(\sigma^R(i_Y \circ \vec{g}f)\right) \circ \varphi(i_X) \ . \end{aligned} \tag{3.9}$$

Rewriting the equality of Display (3.9) twice in the goal shown in Display (3.7) yields the goal

$$g^*\left(\sigma^R(i_Y \circ \vec{g}f)\right)((\varphi(i_X))(x)) = g^*\left(\sigma^R(i_Y \circ \vec{g}f)\right)((\varphi(i_X))(y)) \ \ ,$$

which is true since $g^*\left(\sigma^R(i_Y \circ \vec{g}f)\right)$ is monotone and $(\varphi(i_X))(x) \leq_R (\varphi(i_X))(y)$ by hypothesis. We hence have defined a monad $\hat{\Sigma}_A$ over $\Delta^{\hat{S}}$.

It remains to show that this is a *reduction* monad: for $f \leq f'$, we must prove that $\sigma(f) \leq \sigma(f')$. By Display (3.9), it suffices to show that

$$g^*\left(\sigma^R(i_Y \circ \vec{g}f)\right) \leq g^*\left(\sigma^R(i_Y \circ \vec{g}f')\right)$$

which follows from the fact that R is a reduction monad.

We need a lemma to proceed:

Lemma 3.36. *Given an elementary S–module $V : \mathrm{Model}^{\Delta}(S, \Sigma) \to \mathsf{LRMod}(S, \mathsf{Pre})$ from the category of models of (S, Σ) in S–monads to the large category of modules over S–monads and $x, y \in V(\hat{\Sigma})(X)(t)$, we have*

$$x \leq_A y \in V_X^{\hat{\Sigma}}(t) \quad \Leftrightarrow \quad \forall R : \mathrm{Model}(S, A), \quad V(i_R)(x) \leq_{V_X^R} V(i_R)(y) \ \ ,$$

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

Proof of Lemma 3.36. The proof is done by induction on the derivation of "V elementary". The only interesting case is where $V = M \times N$ is a product:

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

□

Modeling (S, Σ) in $\hat{\Sigma}_A$: For the modeling of types there is nothing to do. Any term arity $s \in \Sigma$ should be modeled by the module morphism $s^{\hat{\Sigma}}$, i.e. by the model of s in $\hat{\Sigma}$. We

have to show that those models are compatible with the preorder A. Given $x \leq_A y$ in $\mathrm{dom}(s, \hat{\Sigma})(X)$, we show (omitting the argument X in $s^{\hat{\Sigma}}(X)(x)$)

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

By definition, we have to show that, for any model R with initial morphism $i = i_R : \hat{\Sigma} \to R$ as before,

$$\varphi(i_X)(s^{\hat{\Sigma}}(x)) \quad \leq_R \quad \varphi(i_X)(s^{\hat{\Sigma}}(y)) \ .$$

But these two sides are precisely the images of x and y under the upper–right composition of the diagram of Definition 3.24 for the morphism of models i_R. By rewriting with this diagram we obtain the goal

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

We know that s^R is monotone, thus it is sufficient to show

$$(\mathrm{dom}(s)(i_R))(x) \quad \leq_R \quad (\mathrm{dom}(s)(i_R))(y) \ .$$

This goal follows from Lemma 3.36 (instantiated for the elementary S–module $\mathrm{dom}(s)$, cf. Definition 3.30) and the hypothesis $x \leq_A y$. We hence have established a model of (S, Σ) in $\hat{\Sigma}_A$. From now on we refer to this model by $\hat{\Sigma}_A$.

$\hat{\Sigma}_A$ satisfies A: The next step is to show that the model $\hat{\Sigma}_A$ satisfies A. Given a rule

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

of A with an elementary S–module V, we must show that for any context $X \in \mathsf{Set}^{\hat{S}}$, any $t \in \hat{S}$ and any $x \in U(\hat{\Sigma}_A)(X)_t$ in the domain of α we have

$$\alpha^{\hat{\Sigma}_A}(x) \quad \leq_A \quad \gamma^{\hat{\Sigma}_A}(x) \ ,$$

where here and later we omit the context argument X and the sort argument t. By Lemma 3.36 the goal is equivalent to

$$\forall R : \mathrm{Model}(\Sigma, A), \quad V(i_R)(\alpha^{\hat{\Sigma}_A}(x)) \quad \leq_{V_X^R} \quad V(i_R)(\gamma^{\hat{\Sigma}_A}(x)) \ . \tag{3.10}$$

Let R be a model of (Σ, A). We continue by proving the statement of Display 3.10 for R. The half-equations α and γ are natural transformations from U to V; since i_R is the carrier of a morphism of (S, Σ)-models from $\Delta\hat{\Sigma}$ to R, we can thus rewrite the goal as

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

which is true since R satisfies A.

Initiality of $\hat{\Sigma}_A$: Given any model R of (Σ, A), the morphism i_R is monotone with respect to the orders on $\hat{\Sigma}_A$ and R by construction of $\leq_A$. It is hence a morphism of models from $\hat{\Sigma}_A$ to R. Uniqueness of the morphisms i_R follows from its uniqueness in the category of models of (S, Σ), i.e. without rules. Hence $(\hat{S}, \hat{\Sigma}_A)$ is the initial object in the category of models of $((S, \Sigma), A)$.

□

In Section 4 we study an example language in detail: we construct the syntax of PCF and show that it yields the initial object in a category of models of PCF. Afterwards, we use the recursion operator obtained from the initiality property to specify a translation from PCF to the untyped lambda calculus. This translation is semantically faithful with respect to the

usual reduction relation of PCF—generated by the rules of Example 3.33—and β-reduction of the lambda calculus.

It is typical for a translation from a high-level language to a low-level language, one reduction step in the source is translated to several reduction steps in the target. Closure under transitivity of our reduction relations is hence crucial for this translation to fit into our framework.

4. A Translation from PCF to ULC via initiality, in Coq

In this section we describe the implementation of the category of models of PCF, equipped with reduction rules, as well as of its initial object. This yields an instance of Theorem 3.35.

Before going into the specifics of this example, we unfold, in the next remark, the initiality property to make explicit the recipe it gives us to specify translations:

Remark 4.1 (Iteration principle by initiality)**.** The universal property of the language generated by a 2–signature yields an *iteration principle* to define maps—translations—on this language, which are certified to be compatible with substitution and reduction in the source and target languages. How does this iteration principle work? More precisely, what data (and proof) needs to be specified in order to define such a translation via initiality from a language, say, $(\hat{S}, \hat{\Sigma}_A)$ to another language $(\hat{S}', \hat{\Sigma}'_{A'})$, generated by signatures (S, Σ, A) and (S', Σ', A'), respectively? The translation is a morphism—an initial one—in the category of models of the signature (S, Σ, A) of the source language. It is obtained by equipping the relative monad $\hat{\Sigma}'_{A'}$ underlying the target language with a model of the signature (S, Σ, A). In more detail:

(1) We give a model of the type signature S in the set $\hat{S}'$. By initiality of $\hat{S}$, this yields a translation $\hat{S} \to \hat{S}'$ of sorts.
(2) Afterwards, we specify a model of the term signature Σ in the monad $\hat{\Sigma}'_{A'}$ by defining suitable (families) of morphisms of $\hat{\Sigma}'_{A'}$–modules. This yields a model R of (S, Σ) in the monad $\hat{\Sigma}'_{A'}$.

By initiality, we obtain a morphism $f : (\hat{S}, \hat{\Sigma}) \to R$ of models of (S, Σ), that is, we obtain a translation from $(\hat{S}, \hat{\Sigma})$ to $(\hat{S}', \hat{\Sigma}')$ as the colax monad morphism underlying the morphism f. However, we have not yet ensured that the translation f is compatible with the respective reduction preorders in the source and target languages.

(3) Finally, we verify that the model R of (S, Σ) satisfies the rules of A, that is, we check whether, for each $\alpha \leq \gamma : U \to V \in A$, and for each context V, each $t \in \hat{S}$ and $x \in U^R_V(t)$,

$$\alpha^R(x) \;\leq\; \gamma^R(x) \;.$$

After verifying that R satisfies the rules of A, the model R is in fact a model of (S, Σ, A). The initial morphism f thus yields a faithful translation from $(\hat{S}, \hat{\Sigma}_A)$ to $(\hat{S}', \hat{\Sigma}'_{A'})$.

To specify a translation from PCF to the untyped lambda calculus, we hence follow the steps outlined in Remark 4.1. However, for the implementation in Coq of this instance, we make several simplifications compared to the general theorem:

- We do not define a notion of 2-signature, but specify directly a Coq type—and a category—of models of PCF with its reduction rules.

- We use dependent Coq types to formalize arities of higher degree (cf. Definition 3.11), instead of relying on modules on pointed categories. A model of an arity of degree n is thus given by a family of module morphisms (of degree zero), indexed n times over the respective object type as described in Lemma 3.16.
- The relation on the initial object is not defined via the formula of Display (3.5), but directly through an inductive type, cf. Code 4.11, and various closures, cf. Code 4.12.

Note that a translation from PCF to the untyped lambda calculus was given already in previous work [Ahr12a]. That translation was between pure syntax, in the sense that no reduction rules were considered on the source and target, and consequently the translation was not asked to be faithful with respect to any reductions. Here, we use the same translation, but upgrade it to a translation between languages equipped with their reduction relations. To make this article self-contained, we repeat many of the code snippets already given in [Ahr12a].

4.1. **Formalization of categories.** In this formalization, categories are formalized as "E-categories"; this means that a category consists of a type, say, O of objects, and a family, say, $A : O \times O \to \mathsf{Setoid}$ of arrows, and operations of identity and composition satisfying the usual laws. Here, a setoid is a type equipped with an equivalence relation denoted $\mathsf{a} == \mathsf{b}$ in our formalization, and the axioms of a category are stated with respect to that equivalence relation, *not* with respect to the identity type of Coq. Setoids as morphisms of a category have been used by Aczel [Acz93] in LEGO (there a setoid is simply called "set") and Huet and Saïbi [HS98] in Coq. A careful analysis of E-categories is given in [Pal17].

When specifying an E-category, one needs to specify, in particular, an equivalence relation on their arrows, that takes the role of equality of arrows. In the category of types and functions of types, a suitable notion of equality of functions is pointwise equality. In the case of a category of functors, a suitable notion of equality on arrows, that is, on natural transformations, is pointwise equality of arrows in the target category. This extends to categories of modules: a suitable notion of equality of module morphisms is pointwise equality of the underlying natural transformations.

Remark 4.2 (E-categories vs. univalent categories)**.** This article was written before univalent foundations became known to the author. At the time of preparing the final version of this article, another approach to categories in type theory has been developed [AKS15]: in univalent foundations, various extensionality principles are available as a consequence of the univalence axiom. This makes it feasible to use of the Martin-Löf identity type for "equality of arrows" in a category.

Going back to the example of categories of functors and natural transformations mentioned above, in univalent foundations one can use propositional extensionality and function extensionality to reduce Martin-Löf identity of natural transformations to pointwise Martin-Löf identity of the underlying maps. One thus recovers the above definition of "sameness" for natural transformations: pointwise sameness of the underlying functions.

This extends to equality between morphisms of monads, and to equality between morphisms of modules: instead of *defining* "sameness" to mean pointwise equality of the underlying functions as is done in the setoid-based approach, one can *show*, in univalent foundations, that Martin-Löf identity is equivalent to pointwise Martin-Löf identity.

For our initiality result this means that both approaches are equivalent, at least in theory. In the formalization it is difficult to judge the difference without actually formalizing both

versions. While formalizers of category theory often speak of "setoid hell" when discussing E-categories, the advantage of having the right notion of "sameness" for morphisms of a category *by definition*—instead of by theorem as in the univalent foundations—might compensate for some of the unwieldiness encountered in the setoid world.

In the following, we use a few Coq notations to make the code look more like pen-and-paper mathematics. This table summarizes some of the notation:

Coq notation	meaning
f ;; g	composition $g \circ f$ in a category (note the differing order)
f == g	setoid equality of morphisms in a category
s ⇒ t	object level (PCF) arrow type
f @ x	object level application (in a model of PCF)

4.2. **Models of PCF.** In this section we explain the formalization of models of PCF with reduction rules (cf. Figures 1 and 2). According to Definitions 3.23 and 3.34, such a model consists of

(1) a model of the types of PCF (in a Coq type U), cf. Example 3.5,
(2) a reduction monad P over the functor Δ^U (in the formalization: IDelta U) and
(3) a model of the arities of PCF (cf. Example 3.21), i.e., morphisms of P–modules with suitable source and target modules such that
(4) the inequalities defining the reduction rules of PCF are satisfied.

A model of PCF hence is a "bundle", i.e. a record type, whose components—or "fields"—are these 4 items. We first define what a model of the term signature of PCF in a monad P is, in the presence of an S_{PCF}–monad (cf. Definition 3.9). Unfolding the definitions, we suppose given a type Sorts, a relative monad P over IDelta Sorts and three operations on Sorts: a binary function Arrow (denoted by an infixed "⇒ ") and two constants Bool and Nat.

```
Variable Sorts : Type.
Variable P : RMonad (IDelta Sorts).
Variable Arrow : Sorts → Sorts → Sorts.
Variable Bool : Sorts.
Variable Nat : Sorts.
Notation "a ⇒ b" := (Arrow a b) (at level 60, right associativity).
```

Here and in the sequel, a short arrow → denotes the type-theoretic function space in Coq. A long arrow ⟶ denotes morphisms in a category; the category at hand will be clear from the context. Equality of morphisms of a category is denoted by ==. Furthermore, we write ≤ to polymorphically denote the relation on a preordered set.

In this context, a model of PCF is given by a bunch of module morphisms satisfying some conditions. We split the definition into smaller pieces, cf. Code 4.3 to 4.7. Note that M[t] denotes the fibre module of module M with respect to t, and d M // u denotes derivation of module M with respect to u. The module denoted by a star ∗ is the terminal module, which is the constant singleton module.

Code 4.3 (1–Signature of PCF)**.**

```
Class PCF_model_struct := {
  app : ∀ u v, (P[u ⇒ v]) x (P[u]) ⟶ P[v]
                        where "A @ B" := (app _ _ _ (A,B));
```

```
abs : ∀ u v, (d P // u)[v] ⟶ P[u ⇒ v];
rec : ∀ t, P[t ⇒ t] ⟶ P[t];
tttt : ∗ ⟶ P[Bool];
ffff : ∗ ⟶ P[Bool];
nats : ∀ m:nat, ∗ ⟶ P[Nat];
Succ : ∗ ⟶ P[Nat ⇒ Nat];
Pred : ∗ ⟶ P[Nat ⇒ Nat];
Zero : ∗ ⟶ P[Nat ⇒ Bool];
CondN: ∗ ⟶ P[Bool ⇒ Nat ⇒ Nat ⇒ Nat];
CondB: ∗ ⟶ P[Bool ⇒ Bool ⇒ Bool ⇒ Bool];
bottom: ∀ t, ∗ ⟶ P[t];
...
```

These module morphisms are subject to some directed equations, specifying the reduction rules of PCF. The β-rule reads as

Code 4.4 (β-rule for models of PCF)**.**

```
beta_red : ∀ r s V y z, abs r s V y @ z ≤ y[∗:= z] ;
...
```

where y[∗:= z] is the substitution of the freshest variable (cf. Definition 2.45) as a special case of simultaneous monadic substitution. The rule for the fixed point operator says that $\mathbf{Y}(f)$ reduces to $f\,(\mathbf{Y}(f))$:

Code 4.5 (Rule for fixpoint operator)**.**

```
Rec_A: ∀ V t g, rec t V g ≤ g @ rec _ _ g
...
```

The other rules concern the arithmetic and logical constants of PCF. Firstly, we have that the conditionals reduce according to the truth value they are applied to:

Code 4.6 (Logical rules of PCF models)**.**

```
CondN_t: ∀ V n m, CondN V tt @ tttt _ tt @ n @ m ≤ n ;
CondN_f: ∀ V n m, CondN V tt @ ffff _ tt @ n @ m ≤ m ;
CondB_t: ∀ V n m, CondB V tt @ tttt _ tt @ n @ m ≤ n ;
CondB_f: ∀ V n m, CondB V tt @ ffff _ tt @ n @ m ≤ m ;
 ...
```

Furthermore, we have that succ(n) reduces to $n+1$ (which in Coq is written S n), reduction of the zero? predicate according to whether its argument is zero or not, and that the predecessor is post–inverse to the successor function:

Code 4.7 (Arithmetic rules of PCF models)**.**

```
Succ_red: ∀ V n, Succ V tt @ nats n _ tt ≤ nats (S n) _ tt ;
Zero_t: ∀ V, Zero V tt @ nats 0 _ tt ≤ tttt _ tt ;
Zero_f: ∀ V n, Zero V tt @ nats (S n) _ tt ≤ ffff _ tt ;
Pred_Succ: ∀ V n, Pred V tt @ (Succ V tt @ nats n _ tt) ≤ nats n _ tt;
Pred_Z: ∀ V, Pred V tt @ nats 0 _ tt ≤ nats 0 _ tt }.
```

After abstracting over the section variables we package all of this into a record type:

```
Record PCF_model := {
  Sorts : Type;
  Arrow : Sorts → Sorts → Sorts;
  Bool : Sorts ;
  Nat : Sorts ;
  pcf_monad :> RMonad (IDelta Sorts);
  pcf_model_struct :> PCF_model_struct pcf_monad Arrow Bool Nat }.
Notation "a ⇒ b" := (Arrow a b) (at level 60, right associativity).
```

The type PCF_model constitutes the type of objects of the category of models of PCF with reduction rules.

4.3. **Morphisms of models.** A morphism of models (cf. Definition 3.24) is built from a morphism g of *type* models and a colax monad morphism over the retyping functor associated to the map g. In the particular case of PCF, a morphism of models from P to R consists of a morphism of models of the types of PCF (with underlying map Sorts_map) and a colax morphism of relative monads which makes commute the diagrams of the form given in Definition 3.24. We first define the diagrams we expect to commute, before packaging everything into a record type of morphisms. The context is given by the following declarations:

```
Variables P R : PCF_model.
Variable Sorts_map : Sorts P → Sorts R.
Hypothesis HArrow : ∀ u v, Sorts_map (u ⇒ v) = Sorts_map u ⇒ Sorts_map v.
Hypothesis HBool : Sorts_map (Bool _ ) = Bool _ .
Hypothesis HNat : Sorts_map (Nat _ ) = Nat _ .
Variable f : colax_RMonad_Hom P R
    (RETYPE (fun t => Sorts_map t))
    (RETYPE_PRE (fun t => Sorts_map t))
  (RT_NT (fun t => Sorts_map t)).
```

Here the colax monad morphism f corresponds to the last component of what we defined to be a colax monad morphism in Definition 2.22, see Remark 2.23.

We explain the commutative diagrams of Definition 3.24 for some of the arities. For the successor arity we ask the following diagram to commute:

Code 4.8 (Commutative diagram for successor arity)**.**

```
Program Definition Succ_hom :=
    Succ ;; f [(Nat ⇒ Nat)] ;; Fib_eq_RMod _ _ ;; IsoPF == ∗⟶ ∗ ;; f ∗∗ Succ.
```

Here the morphism Succ refers to the model of the successor arity either of P (the first occurrence) or R (the second occurrence); Coq is able to figure this out itself. The domain of the successor is given by the terminal module $*$. Accordingly, we have that $\mathrm{dom}(\mathrm{Succ}, f)$ is the trivial module morphism with domain and codomain given by the terminal module. We denote this module morphism by $*\longrightarrow *$. The codomain is given as the fibre of f of type $\iota \Rightarrow \iota$. The two remaining module morphisms are isomorphisms which do not appear in

the informal description. The isomorphism IsoPF is needed to permute fibre with pullback (cf. Lemma 2.44). The morphism Fib_eq_RMod M H takes a module M and a proof H of equality of two object types as arguments, say, H : u = v. Its output is an isomorphism M[u] ⟶ M[v]. Here the proof is of type

Sorts_map (Nat ⇒ Nat) = Sorts_map Nat ⇒ Sorts_map Nat

and Coq is able to figure out the proof itself. The diagram for application uses the product of module morphisms, denoted by an infixed X:

Code 4.9 (Commutative diagram for application arity)**.**

```
Program Definition app_hom' := ∀ u v,
    app u v;; f [( _ )] ;; IsoPF ==
    (f [(u ⇒ v)] ;; Fib_eq_RMod _ (HArrow _ _);; IsoPF ) X (f [(u)] ;; IsoPF ) ;;
       IsoXP ;; f ** (app _ _ ).
```

In addition to the already encountered isomorphism IsoPF we have to insert an isomorphism IsoXP which permutes pullback and product (cf. Lemma 2.42). As a last example, we present the property for the abstraction:

Code 4.10 (Commutative diagram for abstraction arity)**.**

```
Program Definition abs_hom' := ∀ u v,
    abs u v ;; f [( _ )] ==
    DerFib_RMod_Hom _ _ _ ;; IsoPF ;; f ** (abs (_ u) (_ v)) ;; IsoFP ;;
       Fib_eq_RMod _ (eq_sym (HArrow _ _ )) .
```

Here the module morphism DerFib_RMod_Hom f u v corresponds to the morphism

$$\mathrm{dom}(\mathrm{Abs}(u, v), f) = [f^u]_v,$$

and IsoFP permutes fibre with pullback, just like its sibling IsoPF, but the other way round.

We bundle all those properties into a type class:

```
Class PCF_model_Hom_struct := {
  CondB_hom : CondB_hom' ;
  CondN_hom : CondN_hom' ;
  Pred_hom : Pred_hom' ;
  Zero_hom : Zero_hom' ;
  Succ_hom : Succ_hom' ;
  fff_hom : fff_hom' ;
  ttt_hom : ttt_hom' ;
  bottom_hom : bottom_hom' ;
  nats_hom : nats_hom' ;
  app_hom : app_hom' ;
  rec_hom : rec_hom' ;
  abs_hom : abs_hom' }.
```

Similarly to what we did for models, we abstract over the section variables and define a record type of morphisms of models from P to R :

```
Record PCF_model_Hom := {
  Sorts_map : Sorts P → Sorts R ;
  HArrow : ∀ u v, Sorts_map (u ⇒ v) = Sorts_map u ⇒ Sorts_map v;
  HNat : Sorts_map (Nat _ ) = Nat R ;
  HBool : Sorts_map (Bool _ ) = Bool R ;
  model_Hom_monad :> colax_RMonad_Hom P R (RT_NT Sorts_map);
  model_colax_Hom_monad_struct :> PCF_model_Hom_struct
                HArrow HBool HNat model_Hom_monad }.
```

4.4. **Equality of morphisms, category of models.** We have already seen how some definitions that are trivial in informal mathematics, turn into something awful in intensional type theory. Equality of morphisms of models is another such definition. Informally, two such morphisms $a, c : P \to R$ of models are equal if

(1) their map of object types f_a and f_c (Sorts_map) are equal and
(2) their underlying colax morphism of monads—also called a and c—are equal.

In our formalization, the second condition is not even directly expressible, since these monad morphisms do not have the same type: we have, for a context $V \in \mathsf{Set}^P$,

$$a_V : \vec{f_a}(PV) \to R(\vec{f_a}V)$$

and

$$c_V : \vec{f_c}(PV) \to R(\vec{f_c}V) \ .$$

where Set^P is a notation for contexts typed over the set of object types the model P comes with, formally the type Sorts P. We can only compare a_V to c_V by composing each of them with a suitable transport transp again, yielding morphisms

$$R(\mathsf{transp}) \circ a_V : \vec{f_a}(PV) \to R(\vec{f_a}V) \to R(\vec{f_c}V)$$

and

$$c_V \circ \mathsf{transp'} : \vec{f_a}(PV) \to \vec{f_c}(PV) \to R(\vec{f_c}V) \ .$$

As before, for equal fibres $[M]_u$ and $[M]_t$ with $u = t$, the carriers of those transports transp and transp' are terms of the form eq_rect _ _ _ H, where H is a proof term which depends on the proof of

```
∀ x : Sorts P, Sorts_map c x = Sorts_map a x
```

of the first condition. Altogether, the definition of equality of morphisms of models is given by the following inductive proposition:

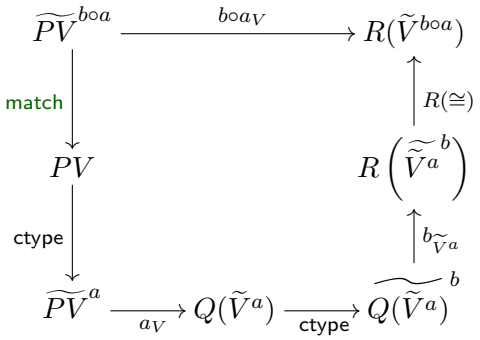
```
Inductive PCF_Hom_eq (P R : PCF_model) : relation (PCF_model_Hom P R) :=
 | eq_model : ∀ a c : PCF_model_Hom P R,
              ∀ H : (∀  t, Sorts_map c t = Sorts_map a t),
              (∀  V, a V ;; rlift R (Transp H V)
                                  ==
                        Transp_ord H (P V) ;; c V ) → PCF_Hom_eq a c.
```

The formal proof that the relation thus defined is an equivalence is inadequately long when compared to its mathematical complexity, due to the transport elimination.

Composition of models is done by composing the underlying maps of sorts, as well as composing the underlying monad morphisms pointwise. Again, this operation, which is trivial from a mathematical point of view, yields a difficulty in the formalization, due to the fact that in Coq, $\vec{g}(\vec{f}V)$ is not convertible to $(g \vec{\circ} f)V$. More precisely, suppose given two morphisms of models $a : P \to Q$ and $b : Q \to R$, given by families of morphisms indexed by V resp. W,

$$a_V : \widetilde{PV}^a \to Q(\widetilde{V}^a) \quad \text{and}$$
$$b_W : \widetilde{QW}^b \to R(\widetilde{W}^b) \ ,$$

where we write $\widetilde{V}^a$ for $\vec{f}_a V$. The monad morphism underlying the composite morphism of models is given by the following definition:



or, in Coq code,

```
Definition comp_model_car : (∀  c : ITYPE U,
        RETYPE (fun t => f' (f t)) (P c) ⟶
     R ((RETYPE (fun t => f' (f t))) c)) :=
  fun (V : ITYPE U) t (y : retype (fun t => f' (f t)) (P V) t) =>
    match y with ctype _ z =>
      lift (M:=R) (double_retype_1 (f:=f) (f':=f') (V:=V)) _
          (b _ _ (ctype (fun t => f' t)
                (a _ _ (ctype (fun t => f t) z ))))
     end.
```

where double_retype_1 denotes the isomorphism in the upper right corner. The proof of the commutative diagrams for the composite monad morphism is lengthy due to the number of arities of the signature of PCF. Definition of the identity morphisms is routine, and in the end we define the category of models of PCF:

```
Program Instance PCF_MODEL :
     Cat_struct (obj := PCF_model) (PCF_model_Hom) := {
  mor_oid P R := eq_model_oid P R ;
  id R := model_id R ;
  comp a b c f g := model_comp f g }.
```

4.5. **One particular model.** We define a particular model, which we later prove to be initial. First of all, the set of object types of PCF is given as follows:

```
Inductive Sorts :=
  | Nat : Sorts
  | Bool : Sorts
  | Arrow : Sorts → Sorts → Sorts.
```

For this section we introduce some notations:

```
Notation "'TY'" := Sorts.
Notation "'IT'" := (ITYPE TY).
Notation "a '~>' b" := (PCF.Arrow a b) (at level 69, right associativity).
```

We specify the set of PCF constants through the following inductive type, indexed by the sorts of PCF:

```
Inductive Consts : TY → Type :=
 | Nats : nat → Consts Nat
 | ttt : Consts Bool
 | fff : Consts Bool
 | succ : Consts (Nat ~> Nat)
 | preds : Consts (Nat ~> Nat)
 | zero : Consts (Nat ~> Bool)
 | condN: Consts (Bool ~> Nat ~> Nat ~> Nat)
 | condB: Consts (Bool ~> Bool ~> Bool ~> Bool).
```

The set family of terms of PCF is given by an inductive family, parametrized by a context V and indexed by object types:

```
Inductive PCF (V: TY → Type) : TY → Type:=
 | Bottom: ∀ t, PCF V t
 | Const : ∀ t, Consts t → PCF V t
 | Var : ∀ t, V t → PCF V t
 | App : ∀ t s, PCF V (s ~> t) → PCF V s → PCF V t
 | Lam : ∀ t s, PCF (opt t V) s → PCF V (t ~> s)
 | Rec : ∀ t, PCF V (t ~> t) → PCF V t.
Notation "a @ b" := (App a b)(at level 43, left associativity).
Notation "M '" := (Const _ M) (at level 15).
```

Monadic substitution is defined recursively on terms:

```
Fixpoint subst (V W: TY → Type)(f: ∀ t, V t → PCF W t)
               (t : TY)(v : PCF V t) : PCF W t :=
  match v with
  | Bottom t => Bottom W t
  | c ' => c '
  | Var t v => f t v
  | u @ v => u >>= f @ v >>= f
  | Lam t s u => Lam (u >>= shift f)
```

```
  | Rec t u => Rec (u >>= f)
  end
where "y >>= f" := (@subst _ _ f _ y).
```

Here, shift f is the substitution map f extended to account for an extended context under the binder Lam. It is equal to the shifted map of Definition 2.35.

Finally, we define a relation on the terms of type PCF via the inductive definition

Code 4.11 (Reduction rules for PCF)**.**

```
Inductive eval (V : IT): ∀ t, relation (PCF V t) :=
 | app_abs : ∀ (s t:TY) (M: PCF (opt s V) t) N,
                  eval (Lam M @ N) (M [∗:= N])
 | condN_t: ∀ n m, eval (condN ' @ ttt ' @ n @ m) n
 | condN_f: ∀ n m, eval (condN ' @ fff ' @ n @ m) m
 | condB_t: ∀ u v, eval (condB ' @ ttt ' @ u @ v) u
 | condB_f: ∀ u v, eval (condB ' @ fff ' @ u @ v) v
 | succ_red: ∀ n, eval (succ ' @ Nats n ') (Nats (S n) ')
 | zero_t: eval ( zero ' @ Nats 0 ') (ttt ')
 | zero_f: ∀ n, eval (zero ' @ Nats (S n)') (fff ')
 | pred_Succ: ∀ n, eval (preds ' @ (succ ' @ Nats n ')) (Nats n ')
 | pred_z: eval (preds ' @ Nats 0 ') (Nats 0 ')
 | rec_a : ∀ t g, eval (Rec g) (g @ (Rec (t:=t) g)).
```

which we then propagate into subterms (cf. Code 4.12) and close with respect to transitivity and reflexivity:

Code 4.12 (Propagation of reductions into subterms)**.**

```
Reserved Notation "x :> y" (at level 70).
Variable rel : ∀ (V:IT) t, relation (PCF V t).
Inductive propag (V: IT) : ∀ t, relation (PCF V t) :=
| relorig : ∀ t (v v': PCF V t), rel v v' → v :> v'
| relApp1: ∀ s t (M M' : PCF V (s ∼> t)) N, M :> M' → M @ N :> M' @ N
| relApp2: ∀ s t (M : PCF V (s ∼> t)) N N', N :> N' → M @ N :> M @ N'
| relLam: ∀ s t (M M':PCF (opt s V) t), M :> M' → Lam M :> Lam M'
| relRec: ∀ t (M M' : PCF V (t ∼> t)), M :> M' → Rec M :> Rec M'
where "x :> y" := (@propag _ _ x y).
```

The data thus defined constitutes a relative monad PCFEM on the functor $\Delta^{T_{\mathsf{PCF}}}$ (IDelta TY). We omit the details.

Now we need to define a suitable morphism (resp. family of morphisms) of PCFEM–modules for any arity (of higher degree). Let α be any such arity, for instance the arity App. We need to verify two things:

(1) we show that the constructor of PCF which corresponds to α is monotone with respect to the order on PCFEM. For instance, we show that for any two terms r s:TY and any V : IDelta TY, the function

```
fun y => App (fst y) (snd y): PCFEM V (r∼>s) x PCFEM V r → PCFEM V s
```

is monotone.

(2) We show that the monadic substitution defined above distributes over the constructor, i.e. we prove that the constructor is the carrier of a *module* morphism.

All of these are very straightforward proofs, resulting in a model PCFE_model of PCF:

```
Program Instance PCFE_model_struct :
        PCF_model_struct PCFEM PCF.arrow PCF.Bool PCF.Nat := {
  app r s := PCFApp r s;
  abs r s := PCFAbs r s;
  rec t := PCFRec t ;
  tttt := PCFconsts ttt ;
  ffff := PCFconsts fff;
  Succ := PCFconsts succ;
  Pred := PCFconsts preds;
  CondN := PCFconsts condN;
  CondB := PCFconsts condB;
  Zero := PCFconsts zero ;
  nats m := PCFconsts (Nats m);
  bottom t := PCFbottom t }.
Definition PCFE_model : PCF_model := Build_PCF_model PCFE_model_struct.
```

Note that in the instance declaration PCFE_model_struct, the Program framework proves automatically the properties of Code 4.4, 4.5, 4.6 and 4.7.

4.6. **Initiality.** In this section we define a morphism of models from PCFE_model to any model R : PCF_model. At first we need to define a map between the underlying sorts, that is, a map Sorts PCFE_model → Sorts R. In short, each PCF type goes to its model in R:

```
Fixpoint Init_Sorts_map (t : Sorts PCFE_model) : Sorts R :=
    match t with
    | PCF.Nat => Nat R
    | PCF.Bool => Bool R
    | u ~> v => (Init_Sorts_map u) ⇒ (Init_Sorts_map v)
    end.
```

The function init is the carrier of what will later be proved to be the initial morphism to the model R. It maps each constructor of PCF recursively to its counterpart in the model R:

```
Fixpoint init V t (v : PCF V t) :
    R (retype (fun t0 => Init_Sorts_map t0) V) (Init_Sorts_map t) :=
  match v with
  | Var t v => rweta R _ _ (ctype _ v)
  | u @ v => app _ _ _ (init u, init v)
  | Lam _ _ v => abs _ _ _ (rlift R
          (@der_comm TY (Sorts R) (fun t => Init_Sorts_map t) _ V ) _ (init v))
  | Rec _ v => rec _ _ (init v)
  | Bottom _ => bottom _ _ tt
  | y ' => match y in Consts t1 return
```

```
    R (retype (fun t2 => Init_Sorts_map t2) V) (Init_Sorts_map t1) with
      | Nats m => nats m _ tt
      | succ => Succ _ tt
      | condN => CondN _ tt
      | condB => CondB _ tt
      | zero => Zero _ tt
      | ttt => tttt _ tt
      | fff => ffff _ tt
      | preds => Pred _ tt
      end
  end.
```

We write i_V for init V and g for Init_Sorts_map. Note that $i_V : \mathsf{PCF}(V) \to g^*\,(R(\vec{g}V))$ really is *the image of the initial morphism under the adjunction* φ of Definition 2.13. Intuitively, passing from init V= i_V to its adjunct $\varphi^{-1}(i_V)$ is done by precomposing with pattern matching on the constructor ctype. We informally denote $\varphi^{-1}(i_V)$ by init V ∘ match.

The map init is compatible with renaming and substitution in PCF and R, respectively, in a sense made precise by the following two lemmas. The first lemma states that, for any morphism $f : V \to W$ in $\mathsf{Set}^{T_{\mathsf{PCF}}}$, the following diagram commutes:

$$\begin{array}{ccc} \mathsf{PCF}(V) & \xrightarrow{\mathsf{PCF}(f)} & \mathsf{PCF}(W) \\ {\scriptstyle \text{init V}}\downarrow & & \downarrow{\scriptstyle \text{init W}} \\ g^*R(\vec{g}V) & \xrightarrow[g^*R(g^*f)]{} & g^*R(\vec{g}W). \end{array}$$

```
Lemma init_lift (V : IT) t (y : PCF V t) W (f : V ⟶ W) :
  init (y //− f) = rlift R (retype_map f) _ (init y).
```

The next commutative diagram concerns substitution; for any $f : V \to \mathsf{PCF}(W)$, the diagram obtained by applying φ to the diagram given in Display (3.8)—i.e. the diagram corresponding to Display (3.9)—commutes:

$$\begin{array}{ccc} \mathsf{PCF}(V) & \xrightarrow{\sigma^{\mathsf{PCF}}(f)} & \mathsf{PCF}(W) \\ {\scriptstyle \text{init V}}\downarrow & & \downarrow{\scriptstyle \text{init W}} \\ g^*R(\tilde{V}) & \xrightarrow[g^*\sigma^R\left(\varphi^{-1}(\text{init W})\circ(g^*f)\right)]{} & g^*R(\tilde{W}). \end{array}$$

In Coq the lemma init_subst proves commutativity of this latter diagram:

```
Lemma init_subst V t (y : PCF V t) W (f : IDelta _ V ⟶ PCFE W):
  init (y >>= f) =
  rkleisli R (SM_ind (fun t v => match v with ctype t p => init (f t p) end))
      _ (init y).
```

This latter lemma establishes almost the commutative diagram for the family $\varphi^{-1}(i_V)$ to constitute a (colax) *monad* morphism, which reads as follows:

$$\begin{array}{ccc} \vec{g}\,(\mathsf{PCF}(V)) & \xrightarrow{\vec{g}\left(\sigma^{\mathsf{PCF}}(f)\right)} & \vec{g}\,(\mathsf{PCF}(W)) \\ {\scriptstyle \mathsf{init\ V} \circ \mathsf{match}}\downarrow & & \downarrow{\scriptstyle \mathsf{init\ W} \circ \mathsf{match}} \\ R(\vec{g}V) & \xrightarrow[\sigma^R(\mathsf{init} \circ \mathsf{match} \circ (\vec{g}f))]{} & R(\vec{g}W). \end{array} \tag{4.1}$$

Before we can actually build a monad morphism with carrier map init V ∘ match, we need to verify that init—and thus its adjunct—is monotone. We do this in 3 steps, corresponding to the 3 steps in which we built up the preorder on the terms of PCF:

(1) the map init is monotone with respect to the relation eval (cf. Code 4.11):

```
Lemma init_eval V t (v v' : PCF V t) : eval v v' → init v ≤ < init v'.
```

(2) the map init is monotone with respect to the propagation into subterms of eval;

```
Lemma init_eval_star V t (y z : PCF V t) : eval_star y z → init y ≤ < init z.
```

(3) the map init is monotone with respect to reflexive and transitive closure of above relation.

```
Lemma init_mono c t (y z : PCFE c t) : y ≤ < z → init y ≤ < init z.
```

We now have all the ingredients to define the initial morphism from PCF to R. As already indicated by the diagram in Display (4.1), its carrier is not given by just the map init, since this map does not have the right type: its domain is given, for any context $V \in \mathsf{Set}^{T_{\mathsf{PCF}}}$, by $\mathsf{PCF}(V)$ and not, as needed, by $\vec{g}\,(\mathsf{PCF}(V))$. We thus precompose with pattern matching in order to pass to its adjunct: for any context V, the carrier of the initial morphism is given by

```
fun t y => match y with
    | ctype _ p => init p
    end
 : retype _ (PCF V) ⟶ R (retype _ W)
```

We recall that the constructor ctype is the carrier of the natural transformation of the same name of Definition 2.13, and that precomposing with pattern matching corresponds to specifying maps on a coproduct via its universal property.
Putting the pieces together, we obtain a morphism of models of PCF:

```
Definition initR : PCF_model_Hom PCFE_model R :=
      Build_PCF_model_Hom initR_s.
```

Uniqueness is proved in the following lemma:

```
Lemma initR_unique : ∀ g : PCFE_model ⟶ R, g == initR.
```

The proof consists of two steps: first, one has to show that the translation of *sorts* coincide. Since the source of this translation is an inductive type—the initial model of the signature of Example 3.5—this proof is done by induction. Afterwards the translations of terms are proved to be equal. The proof is done by induction on terms of PCF. It makes essentially use of the commutative diagrams (cf. Definition 3.24) which we presented for the arities of successor (Code 4.8), application (Code 4.9) and abstraction (Code 4.10). Finally we can declare an instance of Initial for the category PCF_MODEL of models:

```
Instance PCF_initial : Initial PCF_MODEL := {
  Init := PCFE_model ;
  InitMor R := initR R ;
  InitMorUnique R := @initR_unique R }.
```

Checking the axioms used for the proof of initiality (and its dependencies) yields the use of non–dependent functional extensionality (applied to the translations of sorts) and uniqueness of identity proofs, which in the Coq standard library is implemented as a consequence of another, logically equivalent, axiom eq_rect_eq:

```
Print Assumptions PCF_initial.
Axioms:
CatSem.AXIOMS.functional_extensionality : ∀ (A B : Type) (f g : A → B),
                                          (∀  x : A, f x = g x) → f = g
Eq_rect_eq.eq_rect_eq : ∀ (U : Type) (p : U) (Q : U → Type)
                                  (x : Q p) (h : p = p), x = eq_rect p Q x p h
```

4.7. **A model of PCF in the untyped lambda calculus.** We use the iteration principle explained in Remark 4.1 in order to specify a translation from PCF to the untyped lambda calculus which is compatible with reduction in the source and target. According to the principle, it is sufficient to define a model of PCF in the relative monad of the lambda calculus (cf. Example 2.9) and to verify that this model satisfies the PCF rules, formalized in the Coq code snippets 4.4, 4.5, 4.6 and 4.7. The first task, specifying a model of the types of PCF, in the singleton set of types of ULC, is trivial. We furthermore specify models of the term arities of PCF, presented in Code 4.3, by giving an instance of the corresponding type class.

```
Program Instance PCF_ULC_model_s :
 PCF_model_struct (Sorts:=unit) ULCBETAM (fun _ _ => tt) tt tt := {
  app r s := ulc_app r s;
  abs r s := ulc_abs r s;
  rec t := ulc_rec t ;
  tttt := ulc_ttt ;
  ffff := ulc_fff ;
  nats m := ulc_N m ;
  Succ := ulc_succ ;
  CondB := ulc_condb ;
  CondN := ulc_condn ;
  bottom t := ulc_bottom t ;
  Zero := ulc_zero ;
  Pred := ulc_pred }.
```

Before taking a closer look at the module morphisms we specify in order to model the arities of PCF, we note that in the above instance declaration, we have not given the proofs corresponding to code snippets 4.4 to 4.7. Referring back to Remark 4.1, we have not completed the third task, the verification that the given model satisfies the directed equations. The Program feature we use during the above instance declaration is able to

detect that the fields called beta_red, rec_A, etc., are missing, and enters into interactive proof mode to allow us to fill in each of the missing fields.

We now take a look at some of the lambda terms modeling arities of PCF. The carrier of the models of ulc_app is the application of lambda calculus, of course, and similar for ulc_abs. Here the parameters r and s vary over terms of type unit, the type of sorts underlying this model. We use an infixed application and a de Bruijn notation instead of the more abstract notation of nested data types:

```
Notation "a @ b" := (App a b) (at level 42, left associativity).
Notation "'1'" := (Var None) (at level 33).
Notation "'2'" := (Var (Some None)) (at level 24).
```

The truth values **T** and **F** are modeled by

```
Eval compute in ULC_True.
     = Abs (Abs 2)
Eval compute in ULC_False.
     = Abs (Abs 1)
```

Natural numbers are given in Church style, the successor function is given by the term $\lambda nfx.f(n\ f\ x)$. The predecessor is modeled by the constant

$$\lambda nfx.n\ (\lambda gh.h(g\ f))(\lambda u.x)(\lambda u.u),$$

and the test for zero is modeled by $\lambda n.n(\lambda x.F)T$, where F and T are the lambda terms modeling **F** and **T**, respectively.

```
Eval compute in ULC_Nat 0.
     = Abs (Abs 1)
Eval compute in ULC_Nat 2.
     = Abs (Abs (2 @ (Abs (Abs (2 @ (Abs (Abs 1) @ 2 @ 1))) @ 2 @ 1)))
Eval compute in ULC_succ.
     = Abs (Abs (Abs (2 @ (3 @ 2 @ 1))))
Eval compute in ULC_pred.
     = Abs (Abs (Abs (3 @ Abs (Abs (1 @ (2 @ 4))) @ Abs 2 @ Abs 1)))
Eval compute in ULC_zero.
     = Abs (1 @ Abs (Abs (Abs 1)) @ Abs (Abs 2))
```

The conditional is modeled by the lambda term $\lambda pab.p\ a\ b$:

```
Eval compute in ULC_cond.
     = Abs (Abs (Abs (3 @ 2 @ 1)))
```

The constant arity $\perp_A$ is modeled by Ω:

```
Eval compute in ULC_omega.
     = Abs (1 @ 1) @ Abs (1 @ 1)
```

The fixed point operator **Fix** (rec) is modeled by the *Turing* fixed–point combinator, that is, the lambda term

```
Eval compute in ULC_theta.
     = Abs (Abs (1 @ (2 @ 2 @ 1))) @ Abs (Abs (1 @ (2 @ 2 @ 1)))
```

The reason why we use the Turing operator instead of, say, the combinator **Y**,

```
Eval compute in ULC_Y.
    = Abs (Abs (2 @ (1 @ 1)) @ Abs (2 @ (1 @ 1)))
```

is that the latter does not have a property that is crucial for us: we have

$$\Theta(f) \rightsquigarrow^* f\,(\Theta(f))$$

but only

$$\mathbf{Y}(f) \overset{*}{\leftsquigarrow} f\,(\mathbf{Y}(f))$$

via a common reduct. Thus if we would attempt to model the arity rec by the fixed–point combinator $\mathbf{Y}$, we would not be able to prove the condition expressed in Code 4.5. Our more fine-grained approach to operational semantics—using directed equations rather than equations—thus has the drawback of ruling out the translation of the fixpoint construct of PCF to the $\mathbf{Y}$ combinator of the untyped lambda calculus.

As a final remark, we emphasize that while reduction is given as a relation in our formalization, and as such is not computable, the obtained translation from PCF to the untyped lambda calculus is executable in Coq. For instance, we can translate the PCF term negating boolean terms, $\lambda x.\mathsf{cond}_o(x)(\text{false})(\text{true})$, as follows:

Code 4.13 (Computing the translation of boolean negation)**.**

```
Eval compute in
  (PCF_ULC_c ((fun t => False)) tt (ctype _
   (Lam (condB ' @@ x_bool @@ fff ' @@ ttt '')))).
   = Abs (Abs (Abs (Abs (3 @ 2 @ 1))) @ 1 @ Abs (Abs 1) @ Abs (Abs 2))
```

Here the notation "@@" denotes application of PCF, and x_bool is simply a notation for a de Bruijn variable of type Bool of the lowest level, i.e. a variable that is bound by the Lam binder of PCF in above term. The resulting term of the lambda calculus is $\lambda x.(\lambda y.(\lambda z.\lambda w.(w@z@y))@x@(\lambda a.\lambda b.b)@(\lambda a.\lambda b.a))$.

5. Conclusion

We have presented an initiality result for simply-typed languages, based on relative monads from sets to preordered sets.

Our approach is "intrinsically typed", that is, terms are typed from the beginning on—there are no "preterms", and no predicate of well-typedness. This yields a useful recursion principle where preservation of typing under translations is guaranteed by construction.

The modeling of reductions via preorders may be considered too coarse: one term might reduce to another term *in different ways*, but the use of preorders to model reduction does not allow to distinguish two reductions with the same source and target.

Instead of considering *preordered* sets (indexed by sets of free variables) as models of a 2–signature, it would thus be interesting to consider a structure which allows for more fine–grained treatment of reduction, such as graphs or categories. However, one might argue that categories naturally form a bicategory, not a category. Models of a 2–signature would then also form a bicategory, and one might need to adapt some concepts used here to a bicategorical setting.

Acknowledgements. We are very grateful to the anonymous referee for their careful reading and valuable comments, and to the editor Larry Moss for his support.

References

[ACU15] Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. Monads need not be endofunctors. *Logical Methods in Computer Science*, 11(1), 2015.

[Acz93] Peter Aczel. Galois: A Theory Development Project. Technical Report for the 1993 Turin meeting on the Representation of Mathematics in Logical Frameworks, 1993. `http://www.cs.man.ac.uk/~petera/galois.ps.gz`.

[Ahr12a] Benedikt Ahrens. Extended Initiality for Typed Abstract Syntax. *Logical Methods in Computer Science*, 8(2):1 – 35, 2012.

[Ahr12b] Benedikt Ahrens. Initiality for typed syntax and semantics. In Luke Ong and Ruy de Queiroz, editors, *Logic, Language, Information and Computation*, volume 7456 of *Lecture Notes in Computer Science*, pages 127–141. Springer Berlin / Heidelberg, 2012.

[Ahr16] Benedikt Ahrens. Modules over relative monads for syntax and semantics. *Mathematical Structures in Computer Science*, 26:3–37, 2016.

[AKS15] Benedikt Ahrens, Krzysztof Kapulkin, and Michael Shulman. Univalent categories and the Rezk completion. *Mathematical Structures in Computer Science*, 25:1010–1039, 2015.

[AR99] Thorsten Altenkirch and Bernhard Reus. Monadic Presentations of Lambda Terms Using Generalized Inductive Types. In Jörg Flum and Mario Rodríguez-Artalejo, editors, *CSL*, volume 1683 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 1999.

[BB94] Henk Barendregt and Erik Barendsen. Introduction to Lambda Calculus. `ftp://ftp.cs.ru.nl/pub/CompMath.Found/lambda.pdf`, 1994. revised 2000.

[BHKM12] Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. Strongly Typed Term Representations in Coq. *J. Autom. Reasoning*, 49(2):141–159, 2012.

[Bir35] Garrett Birkhoff. On the Structure of Abstract Algebras. In *Proc. Cambridge Phil. Soc.*, volume 31, pages 433–454, 1935.

[BM98] Richard S. Bird and Lambert Meertens. Nested Datatypes. In Johan Jeuring, editor, *LNCS 1422: Proceedings of Mathematics of Program Construction*, pages 52–67, Marstrand, Sweden, June 1998. Springer-Verlag.

[CDT12] The Coq Development Team. *The Coq Reference Manual, version 8.3*, 2012. Available electronically at `https://coq.inria.fr/distrib/8.3pl5/`.

[FG07] Maribel Fernández and Murdoch J. Gabbay. Nominal rewriting. *Information and Computation*, 205(6):917–965, 6 2007.

[FH07] Marcelo P. Fiore and Chung-Kil Hur. Equational Systems and Free Constructions (Extended Abstract). In Lars Arge, Christian Cachin, Tomasz Jurdzinski, and Andrzej Tarlecki, editors, *ICALP*, volume 4596 of *Lecture Notes in Computer Science*, pages 607–618. Springer, 2007.

[Fio02] Marcelo P. Fiore. 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, pages 26–37, New York, NY, USA, 2002. ACM.

[Fio05] Marcelo P. Fiore. Mathematical Models of Computational and Combinatorial Structures. In Vladimiro Sassone, editor, *FoSSaCS*, volume 3441 of *Lecture Notes in Computer Science*, pages 25–46. Springer, 2005.

[FPT99] Marcelo P. Fiore, Gordon Plotkin, and Daniele Turi. Abstract Syntax and Variable Binding. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, LICS '99, pages 193–202, Washington, DC, USA, 1999. IEEE Computer Society.

[GL03] Neil Ghani and Christoph Lüth. Rewriting via coinserters. *Nord. J. Comput.*, 10(4):290–312, 2003.

[GP99] Murdoch J. Gabbay and Andrew M. Pitts. A New Approach to Abstract Syntax Involving Binders. In *14th Annual Symposium on Logic in Computer Science*, pages 214–224, Washington, DC, USA, 1999. IEEE Computer Society Press.

[GP01] Murdoch J. Gabbay and Andrew M. Pitts. A New Approach to Abstract Syntax with Variable Binding. *Formal Aspects of Computing*, 13(3–5):341–363, 2001.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, New York, NY, USA, 1989.

[Hir13] Tom Hirschowitz. Cartesian closed 2-categories and permutation equivalence in higher-order rewriting. *Logical Methods in Computer Science*, 9(3), 2013.

[HM07a] André Hirschowitz and Marco Maggesi. Higher-order theories. 2007. arXiv:0704.2900v2.

[HM07b] André Hirschowitz and Marco Maggesi. Modules over monads and linearity. In Daniel Leivant and Ruy J. G. B. de Queiroz, editors, *WoLLIC*, volume 4576 of *Lecture Notes in Computer Science*, pages 218–237. Springer, 2007.

[HM10] André Hirschowitz and Marco Maggesi. Modules over monads and initial semantics. *Inf. Comput.*, 208(5):545–564, 2010.

[HO00] J. Martin E. Hyland and C.-H. Luke Ong. On full abstraction for PCF: I. Models, observables and the full abstraction problem II. Dialogue games and innocent strategies III. A fully abstract and universal game model. *Information and Computation*, 163:285–408, 2000.

[Hof99] Martin Hofmann. Semantical analysis of higher-order abstract syntax. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, pages 204–213. IEEE Computer Society, 1999.

[HS98] Gérard Huet and Amokrane Saïbi. Constructive Category Theory. In *Proceedings of the Joint CLICS-TYPES Workshop on Categories and Type Theory, Goteborg*. MIT Press, 1998.

[Hur10] Chung-Kil Hur. *Categorical equational systems: algebraic models and equational reasoning*. PhD thesis, University of Cambridge, UK, 2010.

[Lei04] Tom Leinster. *Higher Operads, Higher Categories*. London Mathematical Society Lecture Note Series 298. Cambridge University Press, Cambridge, 2004. arxiv:math.CT/0305049.

[MS03] Marino Miculan and Ivan Scagnetto. A framework for typed HOAS and semantics. In *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden*, pages 184–194. ACM, 2003.

[Pal17] Erik Palmgren. On equality of objects in categories in constructive type theory, 2017. arXiv:1708.01924.

[Pit03] Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.

[Plo77] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, 1977.

[Pow07] John Power. Abstract Syntax: Substitution and Binders. *Electron. Notes Theor. Comput. Sci.*, 173:3–16, 2007.

[TP05] Miki Tanaka and John Power. 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, pages 13–24. ACM, 2005.

[Zsi10] Julianna Zsidó. *Typed Abstract Syntax*. PhD thesis, University of Nice, France, 2010. `http://tel.archives-ouvertes.fr/tel-00535944/`.