Monads from multi-sorted binding signatures*

Benedikt Ahrens[1], Ralph Matthes[2], and Anders Mörtberg[3]

1 **INRIA Rennes Bretagne Atlantique, Nantes, France**
`benedikt.ahrens@inria.fr`
2 **IRIT (CNRS and Université de Toulouse), Toulouse, France**
`ralph.matthes@irit.fr`
3 **Université Côte d'Azur, Inria, Sophia Antipolis, France**
`anders.mortberg@inria.fr`

Abstract

In this work, we provide a construction that yields, from a simple notion of multi-sorted binding signature, the generated "term monad". Here, the monadic multiplication corresponds to a well-behaved substitution operation. To this end, we present a generic construction that associates to any multi-sorted binding signature a signature with strength in the sense of Matthes and Uustalu, in particular, a rank-2 endofunctor.

The signatures thus obtained are the essential ingredient for a categorical construction of a substitution system, which in turn allows us to construct the term monad that is inductively generated by the signature. This construction has been described in previous work and can be reused here with minor modifications.

The original element of this work is the extension of previous results by the present authors for untyped syntax to multi-sorted syntax. The examples in this paper are the syntax of the simply-typed lambda calculus (taking as sorts its types) and the raw syntax of the Calculus of Constructions in the style of Streicher (with only two sorts for types and elements). The whole construction has been rigorously formalized in UniMath, a recent library of univalent mathematics implemented using the Coq theorem prover.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages

Keywords and phrases Variable binding, Categorical semantics, Interactive theorem proving, Monads

Digital Object Identifier 10.4230/LIPIcs.FSCD.2017.

1 Introduction

This work is about a construction of syntax for *multi-sorted* term systems with variable binding, and of a well-behaved substitution operation for that syntax. It generalizes previous work [4] in which untyped syntax and substitution were constructed.

We are working in a type-theoretic meta-theory, specifically given by the UniMath language [13, 14]. Our goal is to construct data types in UniMath that represent the terms of (idealized) functional programming languages, such as the simply-typed lambda calculus. The reasoning principles associated to these data types—recursion and induction principles coming from initiality—can then be used to reason about program behavior.

* This material is based upon work supported by the National Science Foundation under agreement No. DMS-1128155 and CMU 1150129-338510. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. This work has partly been funded by the CoqHoTT ERC Grant 637339.

Usually, e.g., in the proof assistants Coq and Agda, those data types are declared using inductive type generation schemes that are built into the logical kernel of those systems. In UniMath, however, general inductive types are not part of the core of the language in order to ease verification of it. In the present work, we show how to work around this limitation, by constructing a class of data types that represent multi-sorted term systems. The class of term systems under consideration is specified by a notion of multi-sorted binding signature. Our main construction associates, to any such signature, the term monad generated by that signature.

The construction of the term monad proceeds roughly in two steps: in the first step, we construct the underlying functor of the monad, which maps a typed context Γ to the set of well-typed terms, say, $T(\Gamma)$ in that context. Furthermore, we obtain the inclusion of variables into the set of terms, $\eta_\Gamma : \Gamma \to T(\Gamma)$, as the unit of the monad. In the second step, we complement the pair (T, η) to a monad by equipping it with a multiplication operation, using generalized Mendler iteration. The machinery we employ to this end was developed by Matthes and Uustalu [10], formalized in UniMath by Ahrens and Matthes [3], and reworked to be applicable to the sets of UniMath by the authors in [4].

The constructions described in the present work have been formalized in the UniMath language and machine-checked using the Coq proof assistant in which UniMath is implemented. See in particular `https://github.com/UniMath/UniMath/tree/master/UniMath/SubstitutionSystems`.

A brief overview of UniMath and, specifically, its library of category theory that we are relying on, is given next. In Section 2 we give a more detailed overview of previous work we are relying on, and of the work described in the present article.

1.1 About UniMath

The term "UniMath" (Univalent Mathematics) refers to both a core type theory as well as a library of mathematics formalized and computer-checked in this type theory.

The UniMath language features dependent product types, dependent pair types, intensional identity types, and a few base types: empty type, unit type, booleans, and natural numbers. This already allows the construction of the datatype of finite lists over a given type, and we will freely make use of it. UniMath also includes propositional truncation. It furthermore features the univalence axiom, which is not used in full generality in the present work. Some consequences of univalence are, however, used extensively, notably functional and propositional extensionality.

Importantly, the UniMath language does not include arbitrary inductive types. The purpose of this restriction is to keep the "trusted kernel" of the language small, and consequently to ease the construction of semantic models of the language. A significant part of the present work is given by the construction of some "families of inductive types", in the form of initial algebras for rank-2 functors, from the other type constructors in UniMath.

A **proposition** in UniMath is a type in which all elements are identical (i. e., intensionally equal). A **set** is a type for which the identity type is a proposition, in other words a type which satisfies "uniqueness of identity proofs".

The UniMath library contains a significant amount of category theory, for details see [2]. The article [2] calls "precategory" what we here refer to as a category, and reserves the word "category" for precategories with an additional property, called "univalence (for categories)". This property is not relevant for the work reported here.

The category Set has as objects sets and as morphisms from X to Y the set of (type-theoretic) functions from X to Y. Given categories $\mathcal{C}$ and $\mathcal{D}$, we denote by $[\mathcal{C}, \mathcal{D}]$ the category

of functors from $\mathcal{C}$ to $\mathcal{D}$, and natural transformations between them.

1.2 Notational conventions regarding category theory

We assume the reader to be familiar with the concepts of category theory as it is found in the standard text by MacLane [9]. Here, we only point to the specific but rather standard notations and conventions we will follow.

Instead of writing that F is an object of the functor category $[\mathcal{C}, \mathcal{D}]$, we often abbreviate this to $F : [\mathcal{C}, \mathcal{D}]$, but also to $F : \mathcal{C} \to \mathcal{D}$. Given $d : \mathcal{D}$, we call $\underline{d} : \mathcal{C} \to \mathcal{D}$ the functor that is constantly d and 1_d on objects and morphisms, respectively. We write Id for the identity functor on $\mathcal{C}$. These notations hide the category $\mathcal{C}$, which will usually be deducible from the context. We also let *(co)products* denote general indexed (co)products and explicitly write if they are binary.

Categories, functors and natural transformations constitute the prime example of a 2-category. We write $\circ$ for vertical composition of natural transformations and $\cdot$ for their horizontal composition. If one of the arguments to horizontal composition is the identity on some functor, we just write the functor as the respective argument. The corner case where both arguments are the identity on some functors X and Y is just functor composition that is hence written $X \cdot Y$. Horizontal composition of $\mu : F \to G$ and $\nu : F' \to G'$ has $\mu \cdot \nu : F \cdot F' \to G \cdot G'$ provided $F, G : \mathcal{D} \to \mathcal{E}$ and $F', G' : \mathcal{C} \to \mathcal{D}$. The order of vertical composition $\circ$ is the same as of functor composition: if $F, G, H : \mathcal{C} \to \mathcal{D}$ and $\mu : G \to H$ and $\nu : F \to G$, then $\mu \circ \nu : F \to H$ is defined by object-wise composition in $\mathcal{D}$. As a rule of thumb, horizontal composition is used in the nodes in the commutative diagrams, and the diagrams make implicit the vertical composition between the natural transformations described by the edges.

Given a functor $F : [\mathcal{A}, \mathcal{B}]$ and a category $\mathcal{C}$ we define the functor $\_ \cdot F : [\mathcal{B}, \mathcal{C}] \to [\mathcal{A}, \mathcal{C}]$, called "precomposition with F", on functor categories: this functor takes a functor $X : [\mathcal{B}, \mathcal{C}]$ and precomposes it with F, that is, the object mapping is $X \mapsto X \cdot F$, and likewise for the morphisms, i. e., the natural transformations. Analogously, we define the functor $F \cdot \_ : [\mathcal{C}, \mathcal{A}] \to [\mathcal{C}, \mathcal{B}]$ with object mapping $X \mapsto F \cdot X$ and call it "postcomposition with F".

The category $\mathrm{Ptd}(\mathcal{C})$ has, as objects, pointed endofunctors on $\mathcal{C}$, that is, pairs of an endofunctor $F : \mathcal{C} \to \mathcal{C}$ and a natural transformation $\eta : \mathsf{Id} \to F$. We write id for the identity functor with its trivial point. Let U be the forgetful functor from $\mathrm{Ptd}(\mathcal{C})$ to $[\mathcal{C}, \mathcal{C}]$ (that forgets the point). We follow [3] in making explicit the monoidal structure on $[\mathcal{C}, \mathcal{C}]$ that carries over to $\mathrm{Ptd}(\mathcal{C})$: let $\alpha_{X,Y,Z} : X \cdot (Y \cdot Z) \simeq (X \cdot Y) \cdot Z$, $\rho_X : 1_{\mathcal{C}} \cdot X \simeq X$ and $\lambda_X : X \cdot 1_{\mathcal{C}} \simeq X$ denote the monoidal isomorphisms. Notice that all those morphisms are pointwise the identity, but making them explicit is needed for typechecking in the implementation [3]. These isomorphisms make sense when the functors are not endofunctors, and we hence also use the first two for $X : [\mathcal{D}, \mathcal{C}]$, $Y : [\mathcal{E}, \mathcal{D}]$, $Z : [\mathcal{F}, \mathcal{E}]$ and the latter with $X : [\mathcal{C}, \mathcal{D}]$.

2 Background and contents overview

In previous work, [4], we formalized a framework for constructing a monad from a simple notion of binding signature. In this section we discuss this construction and outline the main definitions and results involved in it, for details we refer to the previous work. The steps of the construction are summarized in Figure 1:

Our main goal, in both the previous and present work, is to build up machinery that yields "term monads" for free. By "term monad" we mean a monad T on a suitable category $\mathcal{C}$ such that $T(\Gamma)$ denotes the collection of terms in context Γ. The monadic unit of T then

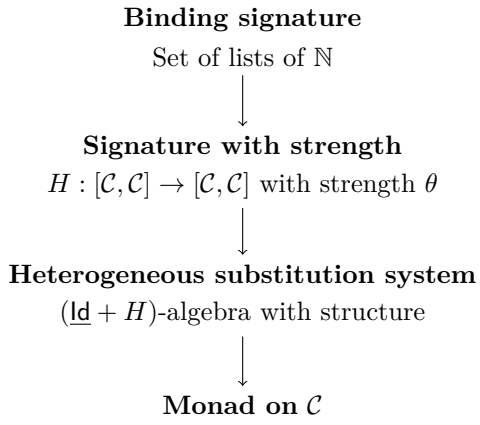

Figure 1 From binding signatures to monads as in [4].

represents the idea that variables of Γ are terms in context Γ, and the monadic multiplication represents the important notion of substitution.

We profit from previous work in the construction of those term monads. Matthes and Uustalu [10] defined a notion of signature, called *signature with strength* below (Definition 1). For initial algebras of such a signature (augmented by a "variable case")—which, intuitively, denote the terms of the language spanned by that signature—they construct a suitable monadic substitution operation via Theorems 3 and 4. This construction proceeds via an intermediate structure, called *substitution system* (Definition 2).

▶ **Definition 1** (Matthes and Uustalu [10])**.** Given a category $\mathcal{C}$, a **signature with strength** is a pair (H,θ) of an endofunctor H on $[\mathcal{C},\mathcal{C}]$, called the signature functor, and a natural transformation $\theta : (H-)\cdot U\sim \to H(-\cdot U\sim)$ between bifunctors $[\mathcal{C},\mathcal{C}]\times \mathrm{Ptd}(\mathcal{C}) \to [\mathcal{C},\mathcal{C}]$ such that θ is "linear" in the second component.

We do not spell out the details of the "linearity" condition since Definition 8 in Section 3.2 captures a more general situation. But the intuition is that given $X : [\mathcal{C},\mathcal{C}]$ and $(Z,e) : \mathrm{Ptd}(\mathcal{C})$ we get a natural transformation $\theta_{X,(Z,e)} : HX\cdot Z \to H(X\cdot Z)$, satisfying suitable equations when (Z,e) is id or of the form $(Z''\cdot Z', e''\cdot e')$ in $\mathrm{Ptd}(\mathcal{C})$.

Since variables play a role that is different from all other term constructors when it comes to substitution, the endofunctor H is not supposed to comprise the inclusion of variables into the terms. The "variable case" is added explicitly to H, by considering the endofunctor $\underline{\mathsf{Id}} + H$ on $[\mathcal{C},\mathcal{C}]$. On objects, it associates with $X : [\mathcal{C},\mathcal{C}]$ the endofunctor $\mathsf{Id} + HX$ on $\mathcal{C}$. Accordingly, an $(\underline{\mathsf{Id}} + H)$-algebra consists of a $T : [\mathcal{C},\mathcal{C}]$ and an $\alpha : \mathsf{Id} + HT \to T$. It is convenient to present α (uniquely) as $[\eta,\tau]$ with $\eta : \mathsf{Id} \to T$ and $\tau : HT \to T$, hence with (T,τ) an H-algebra (but we are not looking for an initial H-algebra).

In order to make sense of this addition of the "variable case", we assume throughout that the base category $\mathcal{C}$ has binary coproducts. This assumption, as well as the choice of $\mathcal{C}$ is left implicit in the statements to follow.

▶ **Definition 2** (Matthes and Uustalu [10])**.** An $(\underline{\mathsf{Id}} + H)$-algebra (T,α) a **heterogeneous substitution system** (HSS) for (H,θ), if, for every $\mathrm{Ptd}(\mathcal{C})$-morphism $f : (Z,e) \to (T,\eta)$, there exists a unique $[\mathcal{C},\mathcal{C}]$-morphism $h : T\cdot Z \to T$, denoted $(\!|f|\!)$, making the diagram in Figure 2 commute.

The two key results in order to obtain a monad from a signature are then:

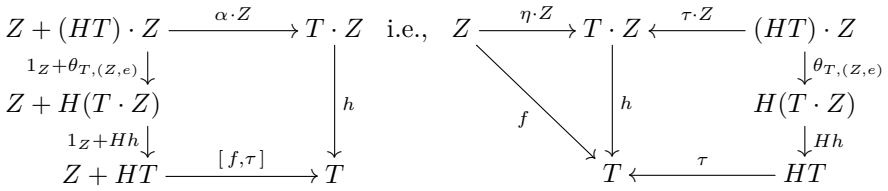

Figure 2 The defining diagram of an HSS.

▶ **Theorem 3** (Construction of an HSS [4]). *Let $\mathcal{C}$ be a category with initial object and colimits of chains, and let (H, θ) be a signature with strength over this category. If H is ω-cocontinuous, then an initial $(\underline{\mathsf{Id}} + H)$-algebra can be constructed and this initial algebra is a heterogeneous substitution system for (H, θ).*

This result requires—unlike the previous work [10, 3]—that the initial algebra comes from ω-cocontinuity of the signature functor (and thus $\mathcal{C}$ has to accommodate that by having an initial object and colimits of chains, while binary coproducts are just our general assumption on $\mathcal{C}$ throughout the paper). The condition on existence of the right adjoint in the mentioned previous variants of the theorem would not allow to apply it to the category Set. Here, we do not need those Kan extensions, but the theorem is less modular.

▶ **Theorem 4** (Matthes and Uustalu [10], formalized in [3]). *Let (H, θ) be a signature with strength. If (T, α) is an HSS for (H, θ), then T, together with the canonically associated $\eta : \mathsf{Id} \to T$ as unit and $(\!|1_{(T,\eta)}|\!) : T \cdot T \to T$ as multiplication, form a monad.*

The relevance of monads for computer science does no longer need to be explained. Monad multiplication as operation of type $T \cdot T \to T$ is rather uncommon there, but an equivalent definition of monads comes with a binding operation instead of monad multiplication, i. e., with an operation bind that "lifts" any morphism $f : \Gamma_2 \to T(\Gamma_1)$ to a morphism $\mathsf{bind}\, f : T(\Gamma_2) \to T(\Gamma_1)$. In the reading of $T(\Gamma)$ as the terms over context Γ, $\mathsf{bind}\, f$ represents parallel substitution of all of the context Γ_2 by terms according to the "substitution rule" f. If the base category $\mathcal{C}$ has binary coproducts, then one can define from bind and η a substitution subst of only part of the context, in particular of one single variable. The general specification of subst is to associate with any $f : \Gamma_2 \to T(\Gamma_1)$ a $\mathsf{subst} f : T(\Gamma_2 \oplus \Gamma_1) \to T(\Gamma_1)$. Also from bind and η, one can define operations weak of weakening the context and exch of exchange of context parts, with the following signatures: $\mathsf{weak} : T(\Gamma_2) \to T(\Gamma_1 \oplus \Gamma_2)$ and $\mathsf{exch} : T(\Gamma_3 \oplus (\Gamma_2 \oplus \Gamma_1)) \to T(\Gamma_2 \oplus (\Gamma_3 \oplus \Gamma_1))$. (Notice that we omit the parameters of all these operations in this discussion.) Then, one can, on the level of that abstract monad, establish a result that corresponds to the interchange law of substitution written in ordinary terms as

$$M[x := N][y := L] = M[y := L][x := N[y := L]]$$

Since no variable names are available, the contexts have to be manipulated to bring the part to be substituted into the right position. The interchange law then becomes for all morphisms $N : \Gamma_3 \to T(\Gamma_2 \oplus \Gamma_1)$ and $L : \Gamma_2 \to T(\Gamma_1)$:

$$(\mathsf{subst}\, L) \circ (\mathsf{subst}\, N) = (\mathsf{subst}((\mathsf{subst}\, L) \circ N)) \circ (\mathsf{subst}(\mathsf{weak} \circ L)) \circ \mathsf{exch}$$

as morphisms $T(\Gamma_3 \oplus (\Gamma_2 \oplus \Gamma_1)) \to T(\Gamma_1)$, describing the effect of the two subsequent substitutions with N, then L. An already interesting special case is with Γ_3 and Γ_2 a terminal object of $\mathcal{C}$ (if it exists). If $\mathcal{C} = \mathsf{Set}$, then this would be substitution of single terms N and L, as in the traditional result mentioned above.

By combining Theorems 3 and 4 one obtains a construction equipping any initial $(\underline{\mathsf{Id}} + H)$-algebra with a monadic substitution operation. What Matthes and Uustalu do not consider is: how to construct those initial algebras for suitable endofunctors H, that can then be used as input to Theorem 3? This question is considered in [4]. The main work in [4] was to construct initial $(\underline{\mathsf{Id}} + H)$-algebras for suitable endofunctors H on $[\mathcal{C}, \mathcal{C}]$, e.g., for the prototypical example of the signature functor Λ specifying the untyped lambda calculus, for objects $X : [\mathcal{C}, \mathcal{C}]$, by

$$\Lambda(X) := X \times X + X \cdot \mathsf{option} \ .$$

Here, assuming $\mathcal{C}$ has a terminal object 1, $\mathsf{option}(A) := 1 + A$ is to be thought of as "extension of a context by a distinguished variable", the variable bound by lambda abstraction.

Not any endofunctor H on $[\mathcal{C}, \mathcal{C}]$ allows for the construction of initial $(\underline{\mathsf{Id}} + H)$-algebras. To delimit a suitable class of functors, *binding signatures* are considered in [4]. The main result of [4] consists in the construction of a signature with strength (H_S, θ_S) (over base category $\mathcal{C}$) to any binding signature S, and a proof that the signature functor H_S is ω-cocontinuous. The proof of ω-cocontinuity constitutes the bulk of the work for [4]. It allows us to use a classical construction, due to Adámek [1], to construct initial $(\underline{\mathsf{Id}} + H_S)$-algebras as colimits of certain diagrams. These algebras are then plugged into the machinery described above to obtain a term monad on Set from a binding signature.

The binding signatures are represented as families of lists of natural numbers, and are hence very simple to specify. For instance, the binding signature specifying the untyped lambda calculus above is given by the family of lists $\{[0, 0], [1]\}$.

In this paper we generalize the notion of binding signature described above to multi-sorted signatures, and provide an analogous construction to that of [4]: to any multi-sorted binding signature we construct a signature with strength over a suitable base category, and show that the underlying functor is ω-cocontinuous.

The suitable base category will be the slice category $\mathsf{Set}/\mathsf{sort}$, where sort is a fixed set of sorts. The steps necessary to generalize the framework of [4] described above to construct term monads on $\mathsf{Set}/\mathsf{sort}$ are hence:

1. Define multi-sorted signatures (next section).
2. Construct an endofunctor H on $[\mathsf{Set}/\mathsf{sort}, \mathsf{Set}/\mathsf{sort}]$ from a multi-sorted signature (Section 3.1).
3. Construct strength θ with laws, for H (Section 3.2).
4. Prove that H is ω-cocontinuous (Section 3.3).
5. Instantiate Theorems 3 and 4 with the constructed data (Section 4).

3 From multi-sorted binding signatures to ω-cocontinuous signatures with strength

In this section we define multi-sorted binding signatures. We then construct a map from multi-sorted binding signatures to signatures in a rather abstract sense, called signatures with strength. Afterwards we construct an initial algebra for any signature with strength obtained from a multi-sorted signature. This involves proving that the signature functor underlying the signature with strength is ω-cocontinuous.

We assume a set sort and define

▶ **Definition 5.** A **multi-sorted binding signature** is given by a set I together with a map $\mathsf{ar} : I \to \mathsf{list}(\mathsf{list}(\mathsf{sort}) \times \mathsf{sort}) \times \mathsf{sort}$.

Intuitively, for any $i : I$, there is a term constructor described by its "arity" $\mathsf{ar}(i)$, whose first component describes the list of arguments: for each argument we specify a list of sorts for the variables bound in it, as well as its sort. The second component of $\mathsf{ar}(i)$ designates the sort of the term constructed by the term constructor pertaining to this index i.

▶ **Example 6.** The standard example is that of simply-typed lambda calculus (STLC),whose representation as a multi-sorted signature is modified from [5]. Assume that sort is closed under a binary operation $\Rightarrow\colon \mathsf{sort} \to \mathsf{sort} \to \mathsf{sort}$ representing function types. We have to put into I the sort parameters of the typing rules of the term constructors of STLC. Thus, I is taken to be $\mathsf{sort} \times \mathsf{sort} + \mathsf{sort} \times \mathsf{sort}$. The left summand pertains to the application operation of STLC while the right summand describes λ-abstraction:

$$\mathsf{ar}(\mathsf{inl}\langle s, t\rangle) := \big\langle [\langle [], s \Rightarrow t\rangle, \langle [], s\rangle], t\big\rangle \qquad \mathsf{ar}(\mathsf{inr}\langle s, t\rangle) := \big\langle [\langle [s], t\rangle], s \Rightarrow t\big\rangle$$

That is, for any sorts s, t, there is an application constructor that takes terms of sorts $s \Rightarrow t$ and s as arguments to yield a term of sort t. And, again for any sorts s, t, there is one λ-abstraction constructor taking one term of sort t with an extra potentially bound variable of sort s, which yields a term of sort $s \Rightarrow t$. This is as close as possible to the usual presentation of the typing rules of STLC.

3.1 The signature functor

Initial algebra semantics for inductive families is given by [7]. However, their formalism is not directly applicable to our needs, because they do not treat large parameters, and their methods do not seem to carry over to large parameters (all types or all sets), since their elements cannot be compared for equality.

We recall that the objects of the slice category $\mathsf{Set}/\mathsf{sort}$ are pairs $(A : \mathsf{Set}, f : A \to \mathsf{sort})$, consisting of a set A of "items" and a typing function f associating sorts with the "items". We will call those objects (A, f) *typing environments* when the set A is viewed as consisting of (object) variable names, hence when f becomes a function associating sorts with variable names. We therefore use letters like Γ to denote objects of $\mathsf{Set}/\mathsf{sort}$. Note that, following common practice, we speak about typing environments instead of sorting environments.

With the aim of an initial algebra semantics for our multi-sorted binding signatures, we are heading for an endofunctor on the functor category $[\mathsf{Set}/\mathsf{sort}, \mathsf{Set}/\mathsf{sort}]$. The rationale is as follows: Given $X : \mathsf{Set}/\mathsf{sort} \to \mathsf{Set}/\mathsf{sort}$, we associate to it an endofunctor $F(X)$ on $\mathsf{Set}/\mathsf{sort}$. An initial algebra for F consists of an object $T : \mathsf{Set}/\mathsf{sort} \to \mathsf{Set}/\mathsf{sort}$ and a(n) (iso)morphism $\mathsf{in} : F(T) \to T$ coding the term constructors. So, the "solution" T for $F(X) \simeq X$ is a semantic representation of the family of multi-sorted terms that depend on typing environments Γ. More in detail, $T(\Gamma)$ consists of a set $\mathsf{Wellsorted}_\Gamma$ of (object) terms and a typing function $\mathsf{sort}_\Gamma : \mathsf{Wellsorted}_\Gamma \to \mathsf{sort}$. It is understood that $\mathsf{Wellsorted}_{(A,f)}$ represents the terms whose free variables are among A and that are typable given the typing function f, and $\mathsf{sort}_{(A,f)}$ yields the (unique) type. Hairsplitting a bit, we should only consider $T(\Gamma)$ as the representation of well-sorted terms in Γ, while $\mathsf{Wellsorted}_\Gamma$ concerns only those that *can be* given a sort.

For any $t : \mathsf{sort}$ we have a projection functor $\mathsf{pr}(t) : \mathsf{Set}/\mathsf{sort} \to \mathsf{Set}$. It maps (A, f) to $\{a \in A \mid f(a) = t\}$. $\mathsf{pr}(t)$ has a left adjoint functor $\hat{t} : \mathsf{Set} \to \mathsf{Set}/\mathsf{sort}$. On objects, it has $\hat{t}(A) = (A, \lambda\_.t)$ (where $\lambda\_.t$ is the function that constantly returns t).

Let 1 denote the terminal set with only element $*$. There is a canonical implementation of a functor $\mathsf{option}(s) : \mathsf{Set}/\mathsf{sort} \to \mathsf{Set}/\mathsf{sort}$ for $s : \mathsf{sort}$, given on objects by

$$\mathsf{option}(s)(A, f : A \to \mathsf{sort}) := (1 + A, [\lambda\_.s, f] : 1 + A \to \mathsf{sort})$$

By the construction of coproducts in slice category $\mathsf{Set}/\mathsf{sort}$ (and more generally), this is

$$\mathsf{option}(s)(A, f) = (1, \lambda\_.s) + (A, f)$$

This can be seen as a generalization of the well-known option functor for any category with binary coproducts and terminal object, where the object mapping is $c \mapsto 1 + c$ to one where 1 is replaced by a fixed object c_0. For $\mathsf{option}(s)$, we take c_0 to be $\mathsf{var}_s := (1, \lambda\_.s)$—a "sorted variable"—in that generalized option functor.

By composition, one obtains $\mathsf{option}(\ell) : \mathsf{Set}/\mathsf{sort} \to \mathsf{Set}/\mathsf{sort}$, for ℓ a list of sorts (note that the order of compositions matters in the implementation, so a choice has been made there).

Let $a = (\ell, t) : \mathsf{list}(\mathsf{sort}) \times \mathsf{sort}$. Define the object part of a functor F^a from $\mathsf{Set}/\mathsf{sort} \to \mathsf{Set}/\mathsf{sort}$ to $\mathsf{Set}/\mathsf{sort} \to \mathsf{Set}$ as:

$$F^a(X) := \mathsf{pr}(t) \cdot X \cdot \mathsf{option}(\ell)$$

That is, F^a itself is the composition of "precomposition with $\mathsf{option}(\ell)$" and "postcomposition with $\mathsf{pr}(t)$". Notice that F^a is not an endofunctor on a category of endofunctors. This will make it necessary, in the next section, to adopt the notion of signature with strength to be able to analyze F^a.

Now, given an arity $(\vec{a}, t)$ with each element of $\vec{a}$ as before and $t : \mathsf{sort}$, define

$$F^{(\vec{a},t)}(X) := (F^{a_1}(X) \times \ldots \times F^{a_m}(X), \lambda\_.t) : \mathsf{Set}/\mathsf{sort} \to \mathsf{Set}/\mathsf{sort}$$

More formally, the endofunctor $F^{(\vec{a},t)}$ on $\mathsf{Set}/\mathsf{sort} \to \mathsf{Set}/\mathsf{sort}$ is obtained by first forming the pointwise product of $F^{a_1}, \ldots, F^{a_n}$, i. e., with

$$X \mapsto (F^{a_1}X \times \ldots \times F^{a_m}X) : \mathsf{Set}/\mathsf{sort} \to \mathsf{Set}$$

on objects, and then by pointwise postcomposition with $\hat{t}$.

▶ **Definition 7.** Given a multi-sorted binding signature (I, ar), its associated signature functor (endofunctor on $[\mathsf{Set}/\mathsf{sort}, \mathsf{Set}/\mathsf{sort}]$) is given by the following pointwise coproduct in $\mathsf{Set}/\mathsf{sort}$:

$$H := \coprod_{i:I} F^{\mathsf{ar}(i)}$$

An initial algebra is later formed for the functor H' that adds the "variable case" to H. For arguments X and $(A, f : A \to \mathsf{sort})$, writing (A', f') for $HX(A, f)$, we want $H'X(A, f)$ to be $(A + A', [f, f'])$, thus want to add to A' all the variable names in $\mathsf{pr}(s)(A, f)$ for any sort s, and adapt f' accordingly. Recall that the binary coproduct in $\mathsf{Set}/\mathsf{sort}$ is given by case analysis, i. e., $(A, f : A \to \mathsf{sort}) + (B, g : B \to \mathsf{sort}) = (A + B, [f, g])$. We thus see that our requirement for H' is an instance of the general principle that one considers the pointwise binary coproduct $H' = \underline{\mathsf{Id}} + H$ for the question of initial algebras.

3.2 Strength and strength laws for the signature functor

The construction of the previous section provides a signature functor for any multi-sorted binding signature. The main result of this section is a construction of a strength for that signature functor, while Section 3.3 contains a proof that the signature functor is ω-cocontinuous.

The extra difficulty mentioned in the previous section is that the construction of the signature functor makes use of functors that are not endofunctors. We cope with this problem by way of the following definition.

▶ **Definition 8** (Pre-signature with strength)**.** Given categories $\mathcal{C}$ and $\mathcal{D}$, a **pre-signature with strength** is a pair (H, θ) of a functor H from $[\mathcal{C}, \mathcal{C}]$ to $[\mathcal{C}, \mathcal{D}]$, called the signature functor, and a natural transformation $\theta : (H-) \cdot U\sim \to H(- \cdot U\sim)$ between bifunctors $[\mathcal{C}, \mathcal{C}] \times \mathrm{Ptd}(\mathcal{C}) \to [\mathcal{C}, \mathcal{D}]$ such that θ is "linear" in the second component.

In detail, the bifunctors applied to a pair of objects $(X,(Z,e))$ with $X : [\mathcal{C},\mathcal{C}]$ and $(Z,e) : \mathrm{Ptd}(\mathcal{C})$ (X for the argument symbolized by $-$ and (Z,e) for the argument symbolized by $\sim$) yield $HX \cdot Z$ and $H(X \cdot Z)$, thus $\theta_{X,(Z,e)} : HX \cdot Z \to H(X \cdot Z)$ in $[\mathcal{C},\mathcal{D}]$. By "linearity" of θ in the second argument we mean the equations

$$\theta_{X,\mathsf{id}} = H(\lambda_X^{-1}) \circ \lambda_{HX}$$

(note that $\lambda_{HX} : HX \cdot 1 \to HX$ and $H(\lambda_X^{-1}) : HX \to H(X \cdot 1)$, using the monoidal isomorphism λ that now has to be used for any functor and not just endofunctors) and

$$\theta_{X,(Z' \cdot Z, e' \cdot e)} = H(\alpha^{-1}_{X,Z',Z}) \circ \theta_{X \cdot Z',(Z,e)} \circ (\theta_{X,(Z',e')} \cdot Z) \circ \alpha_{HX,Z',Z} \ ,$$

as illustrated by the diagram

$$\begin{array}{ccccc}
HX \cdot (Z' \cdot Z) & & \xrightarrow{\quad\theta_{X,(Z' \cdot Z, e' \cdot e)}\quad} & & H(X \cdot (Z' \cdot Z)) \\
\Big\downarrow{\scriptstyle \alpha_{HX,Z',Z}} & & & & \Big\uparrow{\scriptstyle H(\alpha^{-1}_{X,Z',Z})} \\
(HX \cdot Z') \cdot Z & \xrightarrow{\theta_{X,(Z',e')} \cdot Z} & H(X \cdot Z') \cdot Z & \xrightarrow{\theta_{X \cdot Z',(Z,e)}} & H((X \cdot Z') \cdot Z)
\end{array}$$

Definition 1 is then just the case where $\mathcal{C}$ and $\mathcal{D}$ are the same category, whence all functor compositions take place in $[\mathcal{C},\mathcal{C}]$. Thus, the difference w. r. t. the earlier formulation [10] (more precisely the one that makes explicit the monoidal operations [3]) is only that $\mathcal{D}$ need not be identical with $\mathcal{C}$, which in turn asks for a wider interpretation of the monoidal operations. Therefore, the new definition ought rather be seen as a "widening" than as a generalization of the earlier one (the body of the definition is untouched, only some ingredients are more general). It is clear that pre-signatures with strength where $\mathcal{C}$ and $\mathcal{D}$ are not the same category ("heterogeneous pre-signatures with strength") are not meant to form fixed points or heterogeneous substitution systems but only serve as building blocks.

We now sketch the construction of the strength for the obtained signature functor. For $\mathsf{option}(s)$, construct a pointed distributive law [4], compose those distributive laws to obtain one for $\mathsf{option}(\ell)$ and then generate the strength for precomposition with $\mathsf{option}(\ell)$. Pointwise post-composition with a fixed functor (here with $\mathsf{pr}(t)$) generically allows to construct a strength from the given one, which is good for F^a. This is a strength in the heterogeneous sense introduced above in Definition 8.

The product of finitely many such F^{a_i} can be treated easily by using the corresponding construction of strengths for binary products iteratively. Let us mention the implementation detail that the constructed signature with strength ought to have as first component the product of those F^{a_i} w. r. t. convertibility and not just provably. For one construction step, there is no concern, but this has to hold even for the iteration process, so the definitions have to go through the structure twice (as seen in our Coq code).

$F^{(\vec{a},t)}$ is then dealt with by another instance of the strength construction for pointwise postcomposition with a fixed functor, this time $\hat{t}$.

Canonically, the strength carries over to coproducts and thus to the signature functor associated with the given multi-sorted binding signature.

The strength laws are not treated after the strength construction but all those constructions we mention are in our implementation operations on signatures with strength. Hence, following the steps described above, we construct a signature with strength from the given multi-sorted binding signature, having as signature functor the functor H defined in the previous section.

3.3 Proving ω-cocontinuity of the signature functor

In order to prove that the constructed signature functor H is ω-cocontinuous as endofunctor on $[\mathsf{Set}/\mathsf{sort}, \mathsf{Set}/\mathsf{sort}]$, we use the following results about ω-cocontinuous functors that are all proved (sometimes in more general form) in [4]:

▶ **Lemma 9** (Examples of ω-cocontinuous functors)**.**

1. *Any constant functor $\underline{d} : \mathcal{C} \to \mathcal{D}$ is ω-cocontinuous.*
2. *The composition of ω-cocontinuous functors is ω-cocontinuous.*
3. *Let $\mathcal{C}$ and $\mathcal{D}$ be categories with specified binary products and further assume that $d \times -$ is ω-cocontinuous for all $d : \mathcal{D}_0$. The binary product, $F \times G : \mathcal{C} \to \mathcal{D}$, of ω-cocontinuous functors $F, G : \mathcal{C} \to \mathcal{D}$ is ω-cocontinuous.*
4. *Let $\mathcal{C}$ have colimits of chains and let $F : \mathcal{A} \to \mathcal{B}$ be a functor, then the functor $\_ \cdot F : [\mathcal{B}, \mathcal{C}] \to [\mathcal{A}, \mathcal{C}]$ is ω-cocontinuous.*

For point 3 to be applicable (given binary products in the two base categories), it suffices that $\mathcal{D}$ has exponentials; as Set is (locally) cartesian closed (see below) this is fulfilled in our applications. In what follows we will use the fact that $\mathcal{C}/X$ has colimits if $\mathcal{C}$ does and that all kinds of (co)limits lift from $\mathcal{D}$ to the functor category $[\mathcal{C}, \mathcal{D}]$. Proving that Set has colimits is one of the places where we profit from working in univalent type theory. This construction needs set quotients which is something that is not directly available in intensional type theory, for details see [4, Section 3.3].

We also need the following result which is a reformulation of a result in [4]:

▶ **Theorem 10.** *Let $\mathcal{C}$ be a category and $\mathcal{D}$ a category with specified coproducts. Given an I-indexed family of functors $F_i : \mathcal{C} \to \mathcal{D}$ the coproduct $\coprod_{i:I} F_i : \mathcal{C} \to \mathcal{D}$ is ω-cocontinuous.*

This theorem was proved in [4] by decomposing the coproduct into a sequence of functors, this meant that we also needed to assume that the category $\mathcal{C}$ had products. We later learned that this assumption is not necessary and that the above theorem has a direct proof which only requires the assumption that $\mathcal{D}$ has coproducts, so the form of the theorem stated above is the one currently being used in the formalization.

The important facts that are not covered by the above results are ω-cocontinuity of the postcomposition with $\mathsf{pr}(t)$ and the postcomposition with $\hat{t}$. The key lemma for proving these is:

▶ **Lemma 11.** *If $F : \mathcal{C} \to \mathcal{D}$ is a left adjoint then it preserves colimits.*

This standard result was formalized in the previous work and it implies that any left adjoint functor preserves colimits of chains, in other words that it is ω-cocontinuous.

▶ **Lemma 12.** *If $F : \mathcal{A} \to \mathcal{B}$ is a left adjoint then $F \cdot \_ : [\mathcal{C}, \mathcal{A}] \to [\mathcal{C}, \mathcal{B}]$ (postcomposition with F), is a left adjoint.*

Proof. Let $G : \mathcal{B} \to \mathcal{A}$ be the right adjoint to F, i. e., $F \dashv G$. Then $F \cdot \_ \dashv G \cdot \_$. ◀

Hence if F is a left adjoint then postcomposition with it is ω-cocontinuous. As remarked in Section 3.1 the functor $\hat{t}$ is left adjoint to $\mathsf{pr}(t)$ so $\hat{t} \cdot \_$ is ω-cocontinuous. The rest of this section will be devoted to proving that $\mathsf{pr}(t)$ is a left adjoint functor and hence that $\mathsf{pr}(t) \cdot \_$ is ω-cocontinuous.

We first recall the following standard result about the category Set:

▶ **Lemma 13.** *The category Set is locally cartesian closed, that is, Set/X has a terminal object, binary products and exponentials for any set X.*

Proof. This result can easily be seen by the fact that Set/X is equivalent to the functor category $[X, \mathsf{Set}]$ (where X is viewed as a discrete category). As Set is cartesian closed the functor category is also cartesian closed and hence will any slice Set/X be cartesian closed as well. However when working formally it is often easier to work with direct constructions, so we have instead formalized the following direct constructions:

The fact that Set/X has a terminal object follows from the fact that $(X, 1_X)$ is the terminal object in any slice category.

Binary products in slice categories are given by pullbacks in the base category. Hence if we have (Y, f) and (Z, g) in Set/X we get that $(Y, f) \times (Z, g) = \{(y, z) \in Y \times Z \mid f(y) = g(z)\}$.

Finally, to construct the exponential in Set/X we are given (Y, f) and need to construct a right adjoint to $(Y, f) \times -$ in Set/X. Given (Z, g) the exponential object $(Z, g)^{(Y,f)}$ is defined as the set $\{(x, h) \mid x \in X, h : f^{-1}(x) \to g^{-1}(x)\}$ together with the first projection. Checking that this construction is functorial and gives the desired right adjoint is then fairly straightforward. ◀

Let $\mathsf{U} : \mathcal{C}/X \to \mathcal{C}$ be the forgetful functor that we mostly use with $\mathcal{C} = \mathsf{Set}$ and $X = \mathsf{sort}$, and let $(\mathsf{var}_t \times -) : \mathsf{Set}/\mathsf{sort} \to \mathsf{Set}/\mathsf{sort}$ denote the functor that sends $(A, f) : \mathsf{Set}/\mathsf{sort}$ to the product (in $\mathsf{Set}/\mathsf{sort}$) with var_t (recall that this stands for $(1, \lambda\_.t)$). The key observation is then that we get the following isomorphism:

▶ **Lemma 14.** *Given a sort t, the functor $\mathsf{pr}(t) : \mathsf{Set}/\mathsf{sort} \to \mathsf{Set}$ and the composition $\mathsf{U} \cdot ((\mathsf{var}_t) \times -)$ are naturally isomorphic.*

Proof. Given (A, f) in $\mathsf{Set}/\mathsf{sort}$ we easily calculate that $(\mathsf{U} \cdot (\mathsf{var}_t \times -))(A, f) = \{(*, a) \in 1 \times A \mid t = f(a)\}$ which is isomorphic to $\mathsf{pr}(t)(A, f) = \{a \in A \mid f(a) = t\}$. (Recall that we denote by $*$ the element of the terminal set 1.) ◀

To see that $\mathsf{pr}(t)$ is a left adjoint we use the following lemmas:

▶ **Lemma 15.** *If $F : \mathcal{C} \to \mathcal{D}$ is a left adjoint functor that is naturally isomorphic to $G : \mathcal{C} \to \mathcal{D}$, then G is also a left adjoint functor.*

▶ **Lemma 16.** *If $F : \mathcal{A} \to \mathcal{B}$ and $G : \mathcal{B} \to \mathcal{C}$ are both left adjoint functors then $G \cdot F$ is a left adjoint functor.*

▶ **Lemma 17.** *If C has binary products then the forgetful functor $\mathsf{U} : \mathcal{C}/X \to \mathcal{C}$ is a left adjoint functor.*

Proof. The right adjoint to U is the functor that maps objects Y of $\mathcal{C}$ to $(X \times Y, \pi_1) : \mathcal{C}/X$, where π_1 is the first projection. ◀

Combining these results we see that $\mathsf{pr}(t)$ is naturally isomorphic to a functor which is the composition of left adjoints, so $\mathsf{pr}(t)$ is a left adjoint and postcomposition with it is ω-cocontinuous. This means that all of the components of H are ω-cocontinuous and we obtain the main result of the section:

▶ **Theorem 18.** *The signature functor associated to a binding signature is ω-cocontinuous.*

Proof. As H is a coproduct of functors it suffices, by Theorem 10, that each of the $F^{(\vec{a},t)}$ functors are ω-cocontinuous. These are the composition of iterated binary products of F^{a_i} and $\hat{t} \cdot \_$. As we have seen $\hat{t} \cdot \_$ is ω-cocontinuous, so by point 3 of Lemma 9 we only need to check that each F^{a_i} is ω-cocontinuous. We write $a_i = (l, s)$ and note that $F^{(l,s)}$ is the composition of $\_ \cdot \mathsf{option}(l)$ and $\mathsf{pr}(t) \cdot \_$. The first of these is ω-cocontinuous by point 4 of Lemma 9 and the second is ω-cocontinuous by the argument above. ◀

4 From multi-sorted binding signatures to monads, with examples

In Section 3 we defined multi-sorted binding signatures and constructed, to any such signature S over a set sort of sorts, a signature with strength (H_S, θ_S) over the base category $\mathsf{Set}/\mathsf{sort}$. We furthermore showed that the underlying signature functor thus obtained is ω-cocontinuous.

Instantiating Theorems 3 and 4 to with the signatures with strength (H_S, θ_S), we obtain a map associating to any signature S as above the term monad generated by S. Below, we apply this map to some specific examples of multi-sorted binding signatures. We develop these examples, by providing suitable proof principles and one-variable substitution for the syntax thus obtained.

As mentioned in Section 2, the monad for functor T we obtain from Theorem 4 can also be presented with a binding operation bind instead of monad multiplication. Although this may look familiar to functional programmers, it does not properly show how the sorts are handled through the use of the slice category. Recall that, for any typing environment Γ, we write $T(\Gamma)$ as $(\mathsf{Wellsorted}_\Gamma, \mathsf{sort}_\Gamma)$. We can define, for typing environments (A_1, f_1) and Γ_2, an operation $\mathsf{bindslice}$ that takes as arguments a function $f : A_1 \to \mathsf{Wellsorted}_{\Gamma_2}$, the hypothesis (called H) $\forall a_1 : A_1.\mathsf{sort}_{\Gamma_2}(f\, a_1) = f_1\, a_1$ and a(n object) term $M : \mathsf{Wellsorted}_{(A_1,f_1)}$ and yields a result of type $\mathsf{Wellsorted}_{\Gamma_2}$, representing the term obtained from M by substituting all free variables (taken from set A_1) in parallel, according to the "substitution rule" f. By construction, it fulfills $\mathsf{sort}_{\Gamma_2}(\mathsf{bindslice}\, f\, H\, M) = \mathsf{sort}_{(A_1,f_1)} M$.

With little extra work, we define from $\mathsf{bindslice}$ an operation $\mathsf{substslice}$ that represents substitution of just one distinguished variable in a term M by a term N of the same sort and gets by construction that $\mathsf{substslice}$ preserves the sort of the term M. This operation $\mathsf{substslice}$ is a "user-friendly" version of the instance of generic subst for $\mathsf{Set}/\mathsf{sort}$ that associates with a morphism $N : \mathsf{var}_t \to T(\Gamma)$ a morphism $\mathsf{subst}\, N : T(\mathsf{option}(t)(\Gamma)) \to T(\Gamma)$, since such N are in bijection with terms of sort t in Γ, by definition of the slice category. We also directly get the interchange law of substitution as an instance, but its interpretation in terms of the "user-friendly" versions of all the operations has not been completed at the time of writing.

We may conclude by saying that the use of the slice category ensures that substitution properly respects the sorts, and that the signatures with strength control proper behavior of substitution for the object term constructors with respect to (typed) binding.

We finally instantiate the construction we just obtained, by way of two examples: STLC, and the Calculus of Constructions as presented by Streicher [11].

4.1 First example: simply-typed lambda calculus

As explained in Example 6, STLC can be represented as a multi-sorted binding signature by assuming that we have a set sort closed under a binary operation $\Rightarrow$: $\mathsf{sort} \to \mathsf{sort} \to \mathsf{sort}$. We then let $I = \mathsf{sort} \times \mathsf{sort} + \mathsf{sort} \times \mathsf{sort}$ and define

$$\mathsf{ar}(\mathsf{inl}\langle s, t\rangle) := \big\langle [\langle [], s \Rightarrow t\rangle, \langle [], s\rangle], t \big\rangle$$
$$\mathsf{ar}(\mathsf{inr}\langle s, t\rangle) := \big\langle [\langle [s], t\rangle], s \Rightarrow t \big\rangle$$

We apply the construction described in Section 3.1 and obtain a functor Λ which is the coproduct of two families of functors Λ_{app} and Λ_λ indexed by sorts s and t. These functors are defined pointwise at $X : [\mathsf{Set}/\mathsf{sort}, \mathsf{Set}/\mathsf{sort}]$ as:

$$\Lambda_{\mathsf{app}}(X) = \hat{t} \cdot (\mathsf{pr}(s \Rightarrow t) \cdot X \times \mathsf{pr}(s) \cdot X)$$
$$\Lambda_\lambda(X) = \widehat{s \Rightarrow t} \cdot (\mathsf{pr}(t) \cdot X \cdot \mathsf{option}(s))$$

In order to obtain exactly these functors in the formalization special care has to be taken for the base cases. For example when defining $\mathsf{option}(\ell)$ for a list of sorts one has to be careful to not compose with the identity functor in the base case as $\mathsf{Id} \cdot F$ is not judgmentally equal to F. Instead one has to do case analysis on whether the list is empty and only if this is the case return the identity functor. These subtle details then makes it a little less direct to formalize the construction of the strength laws and the ω-cocontinuity proof compared to the paper, for details see the formalization.

Once we have Λ we can then compute the initial algebra of $\underline{\mathsf{Id}} + \Lambda$ which is a functor $L : \mathsf{Set}/\mathsf{sort} \to \mathsf{Set}/\mathsf{sort}$ together with $\alpha : \mathsf{Id} + \Lambda(L) \to L$. The left summand gives a constructor for variables, $\mathsf{var} : \mathsf{Id} \to L$, and the right summand gives constructors for application and abstraction indexed by sorts s and t:

$$\mathsf{app} : \Lambda_{\mathsf{app}}(L) \to L \qquad\qquad \mathsf{lam} : \Lambda_{\lambda}(L) \to L$$

Initiality of $\underline{\mathsf{Id}} + \Lambda$ then gives us the recursion principle for the simply-typed lambda calculus as a catamorphism.

We have defined all of this in UniMath, but must omit it due to space constraints, the interested reader is encouraged to consult the formalization. We want to emphasize that it is quite convenient to use this framework, for instance, we can easily define the binding signature `STLC_sig` and directly obtain a monad on $\mathsf{Set}/\mathsf{sort}$ by:

```
Definition STLC_Monad : Monad (SET / sort) :=
  MultiSortedSigToMonad sort STLC_Sig.
```

This could then be used to animate STLC—as mentioned before, the abstract monad view is only a part of the picture since it does not say anything in particular about the behavior of substitution w. r. t. the term constructors.

4.2 Second example: calculus of constructions à la Streicher

In [11] there is a short definition of the syntax of the (impredicative) Calculus of Constructions [6] which we henceforth call CoC. Below is a slight variation of this syntax:

A, B	$::=$	$\Pi(A, x.B)$	Product of types
	$\mid$	$Prop$	Type of propositions
	$\mid$	$Proof(t)$	Type of proofs of proposition t
t, u	$::=$	x	Variable
	$\mid$	$\lambda(A, x.t)$	Function abstraction
	$\mid$	$App(A, x.B, t, u)$	Function application
	$\mid$	$\forall(A, x.t)$	Universal quantification over propositions t

This is a simultaneous definition of the two syntactic classes of *types* (A, B, ...) and *elements* (t, u, ...). The syntax $x.B$ and $x.t$ should be read as x is bound in B and t, respectively. Note that the function application is annotated with its types, this is necessary for proving initiality of the term model constructed from this syntax. Note also that the description of *Proof(t)* and universal quantification mentions that t ought to be a proposition, but this will only be expressed in the typing system for the CoC.

With the aim of using the sort system to keep these syntactic categories properly apart, we associate sorts with each of them, so in this example we call them ty for types and el for elements. We will hence construct a 2-sorted signature where $\mathsf{sort} = \{\mathsf{ty}, \mathsf{el}\}$. As

there are 6 constructors except for the variable that does not have to be specified, we let $I = \{\mathsf{prd}, \mathsf{prp}, \mathsf{prf}, \mathsf{lam}, \mathsf{app}, \mathsf{all}\}$ and define:

$$\begin{array}{lll} \mathsf{ar}(\mathsf{prd}) & := \langle [\langle [], \mathsf{ty}\rangle, \langle [\mathsf{el}], \mathsf{ty}\rangle], \mathsf{ty}\rangle & \mathsf{ar}(\mathsf{lam}) := \langle [\langle [], \mathsf{ty}\rangle, \langle [\mathsf{el}], \mathsf{el}\rangle], \mathsf{el}\rangle \\ \mathsf{ar}(\mathsf{prp}) & := \langle [], \mathsf{ty}\rangle & \mathsf{ar}(\mathsf{app}) := \langle [\langle [], \mathsf{ty}\rangle, \langle [\mathsf{el}], \mathsf{ty}\rangle, \langle [], \mathsf{el}\rangle, \langle [], \mathsf{el}\rangle], \mathsf{el}\rangle \\ \mathsf{ar}(\mathsf{prf}) & := \langle [\langle [], \mathsf{ty}\rangle], \mathsf{ty}\rangle & \mathsf{ar}(\mathsf{all}) := \langle [\langle [], \mathsf{ty}\rangle, \langle [\mathsf{el}], \mathsf{el}\rangle], \mathsf{el}\rangle \end{array}$$

In the formalization, we can just use the standard set with six elements and its recursor for definition by cases on the elements, bool for sort, true for ty and false for el. The constructions can be instantiated and yield another monad on Set/sort, for the raw syntax of the CoC. We can also extract the constructors and obtain a recursion principle as a catamorphism.

Notice that the sorts have another function in this example than before for the STLC. The sorts in the STLC example represented the types of the language being modeled, and so we obtained the typed terms of STLC. In the present example, however, the sorts take care of organizing a simultaneous inductive definition of expressions involving binding and only guarantee the well-formedness according to the grammar of the raw syntax; we thus obtain the raw syntax of the CoC.

5 Conclusions and future work

Conclusions: We have shown how to extend the construction of monads for the term systems described by simple binding signatures to those involving sorts. For their description, we propose a (still simple) notion of multi-sorted binding signatures. Any such multi-sorted binding signature is then canonically associated with a signature functor. While the definition goes smoothly, it shows the need to leave the setting of endofunctors on endofunctor categories, in particular, we have to go back and forth between Set/sort and Set, by the adjunction $\hat{t} \dashv \mathsf{pr}(t)$ for any sort t. To validate that definition, we construct corresponding operations on signatures with strength. Since there are different categories involved, this requires the (rather simple) extension to pre-signatures with strength. For the existence of initial algebras and also the subsequent construction of heterogeneous substitution systems, we established ω-cocontinuity of the signature functor. Again, this requires to establish preservation of ω-cocontinuity by all the operations involved in the definition of the signature functor. By exploiting another adjunction between Set/sort and Set with the forgetful functor as left adjoint, we obtain even a right adjoint for $\mathsf{pr}(t)$, and having a right adjoint is a sufficient (and high-level) criterion for being ω-cocontinuous.

With all these specific preparations in place, the generality of the previously obtained constructions eventually pays off: the substitution operation and the monad structure follow by instantiating those former constructions to the slice category Set/sort as base category. So, while in the end the results are set-theoretic in nature (although given by a type-theoretic formalization in univalent foundations), it is crucial that already the former work had been organized (structured and adjusted concerning the level of abstraction) through category theory. Only in this sense, the multi-sorted case becomes an instance of the unsorted one: the signatures with strength obtained for the simple binding signatures were the basis of a construction of a monadic substitution operation with an abstract category $\mathcal{C}$, although $\mathcal{C} = \mathsf{Set}$ was intended, but the signatures with strength we construct from multi-sorted binding signatures in the main technical contribution of this article serve equally well for the generic constructions, by choosing $\mathcal{C} = \mathsf{Set/sort}$.

This work has been entirely formalized in UniMath, see in particular `https://github.com/UniMath/UniMath/tree/master/UniMath/SubstitutionSystems`, where also our examples of the previous section can be consulted.

Future work: In the future, we would like to investigate the following questions:

- Can we generalize and work with $[\mathsf{sort}, \mathcal{C}]$ for category $\mathcal{C}$ with extra structure instead? What structure is $\mathcal{C}$ required to have?
- The monad we construct from a binding signature is initial in a suitable category; see, e.g., [8]. To show that, we need to formalize the notion of module over a monad, and constructions on those modules. This work is underway.
- Voevodsky [12] constructs a C-system from a module over a monad on sets. Such a pair can be obtained from a "2-sorted monad", e.g., from the monad of CoC presented in Section 4.2. We would like to formalize Voevodsky's construction and apply it to our monads.

Acknowledgements. We are grateful to Vladimir Voevodsky for many discussions on the subjects of multi-sorted term systems and the intended applications to type theory. We thank Peter LeFanu Lumsdaine for suggesting and formalizing an improvement to one of our proofs.

References

1 Jiří Adámek. Free algebras and automata realizations in the language of categories. *Commentationes Mathematicae Universitatis Carolinae*, 15(4):589–602, 1974.
2 Benedikt Ahrens, Krzysztof Kapulkin, and Michael Shulman. Univalent categories and the Rezk completion. *Math. Struct. in Comp. Science*, 25:1010–1039, 2015. `arXiv:1303.0584`.
3 Benedikt Ahrens and Ralph Matthes. Heterogeneous substitution systems revisited. *ArXiv e-prints*, 2016. `http://arxiv.org/abs/1601.04299`.
4 Benedikt Ahrens, Ralph Matthes, and Anders Mörtberg. From signatures to monads in UniMath. *ArXiv e-prints*, 2016. `https://arxiv.org/abs/1612.00693`.
5 Benedikt Ahrens and Julianna Zsidó. Initial Semantics for higher–order typed syntax in Coq. *Journal of Formalized Reasoning*, 4(1):25–69, September 2011.
6 Thierry Coquand and Gérard Huet. The Calculus of Constructions. Technical Report RR-0530, INRIA, May 1986. URL: `http://hal.inria.fr/inria-00076024`.
7 Makoto Hamana and Marcelo P. Fiore. A foundation for GADTs and inductive families: dependent polynomial functor approach. In Jaakko Järvi and Shin-Cheng Mu, editors, *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming, WGP@ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 59–70. ACM, 2011.
8 André Hirschowitz and Marco Maggesi. Modules over monads and initial semantics. *Inf. Comput.*, 208(5):545–564, 2010.
9 Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, second edition, 1998.
10 Ralph Matthes and Tarmo Uustalu. Substitution in non-wellfounded syntax with variable binding. *Theoretical Computer Science*, 327(1-2):155–174, 2004.
11 Thomas Streicher. *Semantics of Type Theory*. Progress in Theoretical Computer Science. Birkhäuser Basel, 1991.
12 Vladimir Voevodsky. C-system of a module over a monad on sets. *ArXiv e-prints*, 2014. `http://arxiv.org/abs/1407.3394`.
13 Vladimir Voevodsky. An experimental library of formalized mathematics based on the univalent foundations. *Mathematical Structures in Comp. Sci.*, 25:1278–1294, 6 2015.
14 Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. *UniMath*: Univalent Mathematics. Available at `https://github.com/UniMath`.