HETEROGENEOUS SUBSTITUTION SYSTEMS REVISITED

BENEDIKT AHRENS AND RALPH MATTHES

Abstract. Matthes and Uustalu (TCS 327(1–2):155–174, 2004) presented a categorical description of substitution systems capable of capturing syntax involving binding which is independent of whether the syntax is made up from least or greatest fixed points. We extend this work in two directions: we continue the analysis by creating more categorical structure, in particular by organizing substitution systems into a category and studying its properties, and we develop the proofs of the results of the cited paper and our new ones in UniMath, a recent library of univalent mathematics formalized in the Coq theorem prover.

CONTENTS

1. Introduction

Given a first-order signature over some supply of variables, substitution is nearly a homomorphism: the substitution function commutes with all term-forming operations (however, at leaf positions, variables may get replaced by terms). But substitution also gives rise to a monad structure. For this, it is useful to see the variable supply of the terms as a parameter: writing TA for the set of terms over variable supply A (those variables that may occur free in the terms), parallel substitution associates with each substitution rule f, which is a function from A to TB, a substitution function $[f] : TA \to TB$, and for a given term $t : TA$, the term $t[f] : TB$ (notice the post-fix notation for function $[f]$) is the result of the parallel substitution that replaces each occurrence of a variable $x : A$ in t by $fx : TB$. In fact, the function T, the function that injects variables into terms, and the operation of parallel substitution together form a monad in the format of a Kleisli triple over the category of sets and functions. Notice that the types serve as a means of tracking the (names of) variables that may occur free in a term, the object syntax itself is untyped. The parameter A plays a more prominent role as soon as variable binding is allowed in the object syntax: for pure λ-calculus, bound and free variable occurrences have to be distinguished, and even the constructors of the object language relate terms with different variable supply, in particular λ-abstraction assumes an argument term where the newly bound variable is added to the variable supply (this will be seen with more details in Section 8.). Although parallel substitution $t[f]$ has to be defined with extra care to avoid capture of free variables of some fx by binders in t, it is still (modulo α-equivalence) nearly a homomorphism, and it still yields a monad [10]. However, the monad laws by themselves do not express the (nearly) "homomorphic nature" of substitution.

In previous work, Matthes and Uustalu [23] define a notion of "heterogeneous substitution system", the purpose of which is to axiomatize substitution and its desired properties. Such a substitution system is given by an algebra of a signature functor, equipped with an operation—which is to be thought of as substitution— that is compatible with the algebra structure map in a suitable sense. The term "heterogeneous" refers to the fact that the underlying notion of signature encompasses variable binding constructions and also explicit substitution a. k. a. flattening. More precisely, the signature is based on a rank-2 functor H (an endofunctor on a category of endofunctors) for the respective domain-specific signature, to which a monadic unit is explicitly added. The latter corresponds to the inclusion of variables into the elements that are considered as terms (in a quite general sense) over their variable supply. The name "rank-2 functor" stems from the rank of the type operator that transforms type transformations into type transformations—hence has kind $(\mathbf{Set} \to \mathbf{Set}) \to (\mathbf{Set} \to \mathbf{Set})$—which may be seen as backbone of H in case the base category is $\mathbf{Set}$. In this rank-2 setting, the carrier of the algebra is an endofunctor, and since a monadic unit is already present, a natural question is if one obtains a monad. In that paper, it is then shown that for any heterogeneous substitution system this is indeed the case; multiplication of the monad is derived from the "substitution" operation which is parameterized by a morphism f of pointed endofunctors and consists in asking for a unique solution that makes a certain diagram commute. Monad multiplication and one of the monad laws is obtained from the existence of a solution in the case that f is the identity, while the other monad laws are derived from uniqueness for two other choices of f.

Furthermore, it is shown there that "substitution is for free" for both initial algebras as well as—maybe more surprisingly—for (the inverse of) final coalgebras: if the initial algebra, resp. terminal coalgebra, of a given signature functor exists, then it, resp. its inverse, can be augmented to a substitution system (for the former case, and in order to easily use generalized iteration [13], it is assumed that the functor $- \cdot Z$ has a right adjoint for every endofunctor Z). Indeed, it was one of the design goals of the axiomatic framework of heterogeneous substitution systems to be applicable to *non-wellfounded* syntax as well as to wellfounded syntax, whereas related work (e.g., [15, 5]) frequently only applies to wellfounded syntax.

Examples of substitution systems are thus given by the lambda calculus, with and without explicit flattening, but also by languages involving typing and *infinite* terms.

The goal of the present work is twofold:

Firstly, we extend the work by Matthes and Uustalu [23]; in particular, we introduce a natural notion of *morphisms* of heterogeneous substitution systems, thus arranging them into a category. We then show that the construction of a monad from a heterogeneous substitution system from [23] extends functorially to morphisms. Moreover, we prove that the substitution system obtained in [23] by equipping the initial algebra with a substitution operation, is initial in the corresponding category of substitution systems. This makes use of a general fusion law for generalized iteration [13]. Moreover, we prove that the property of being initial in the category of algebras lifts to initiality of the associated substitution system in the corresponding category. As an example of the usefulness of our results, we express the resolution of explicit flattening of the lambda calculus as a(n initial) morphism of substitution systems.

A second part of our work is the formalization of some of our results in univalent foundations, more specifically, building upon the `UniMath` library [1]. This basis of our formalization is suitable in that it provides extensionality (functional and propositional) in a natural way and hereby avoids the use of setoids that would otherwise be inevitable; indeed, since our results are not about categories *in abstracto* but use general categorical concepts in more concrete instances such as the endofunctor category over a given category or its extension by a "point", we need extensionality axioms for the instantiation. We profit from the existing category theory library [7] in `UniMath`.

Related work. Related work is extensively discussed in Matthes and Uustalu's article [23].

In the meantime, monads and modules over monads, have been used by Hirschowitz and Maggesi [16, 17] to define models of syntax, and to give a categorical characterization thereof.

The notion of signature introduced in [23] and formalized in the present work is similar to that employed in Hirschowitz and Maggesi's most recent work [18]. One difference is that we do not, in the present work, insist on our signature functor to be ω-cocontinuous, since we do not worry about the existence of initial algebras, but assume them to exist. In our follow-up work with Mörtberg [8] on the construction of initial algebras in sets, however, this condition will be of the essence.

Monads and modules over monads can also be used as the basis, the "raw syntax", from which dependently typed theories are carved out, as exhibited by Voevodsky

[30]. Our formalization provides one of the many steps involved, providing a monad structure on an initial algebra of a rank-2 endofunctor.

Synopsis. In Section 2 we first give a brief overview of the univalent foundations we work in. Afterwards, we review the definition of categories in those foundations, and finally, we show how the foundations are realized in the proof assistant Coq.

In Section 3 we define a few basic concepts and introduce notation.

In Section 4 we present "Generalized Iteration in Mendler-style", and a fusion law satisfied by this form of iteration. The presented results will be used in Section 7.

In Section 5 we review the notion of heterogeneous substitution system. Afterwards, we define a category of substitution systems and prove a few properties about that category.

In Section 6 we state one of the main results of [23], the construction of a monad from a substitution system. We then prove that the map thus constructed extends to morphisms and yields a faithful functor.

In Section 7 we state another of the important results of [23]: the construction of a substitution system from an initial algebra via Generalized Iteration in Mendler-style as presented in Section 4. We show that the obtained substitution system is again initial, using the fusion law stated in 4.

In Section 8, we construct a particular morphism of substitution systems, the underlying map of which "computes away" explicit substitution of lambda calculus.

Most of the results presented in this article, both by Matthes and Uustalu [23] and our new results, have been formalized, based on the `UniMath` library [1]. More precisely, all results except for Theorem 20 and Lemmas 23 and 19 are proved in our formalization; Section 9 provides some technical details about our library.

2. Univalent mathematics

The original article [23] is written without referring to a specific foundation of mathematics. Indeed, the authors use purely categorical methods to derive their results.

Our analysis and continuation of that article takes place in a *type-theoretic* foundation, more specifically, in a type theory augmented by Voevodsky's *Univalence Axiom*. The resulting theory, to which we refer by the name "HoTT" in this article, is extensively described elsewhere [29]; we do not attempt to give a comprehensive introduction to HoTT or to the Univalence Axiom in this article. Instead, here we focus on some of the salient features of HoTT and indicate why they are important to us.

2.1. **About univalent foundations.** By "univalent foundations" we refer to an intensional Martin-Löf type theory (IMLTT) augmented by Voevodsky's univalence axiom. In the following, we give a brief overview of the type constructors available in univalent foundations, and a technical statement of the univalence axiom.

Technically, the univalent foundation we work in is a dependent type theory. For a dependent type B over A, written $x : A \vdash B(x)$, there is the **dependent sum** $\sum_{(x:A)} B(x)$, elements of which are dependent pairs (a, p) where $a : A$ and $p : B(a)$. The type $\prod_{(x:A)} B(x)$ is the type of dependent functions from A to B, that is, a function $f : \prod_{(x:A)} B(x)$ maps $a : A$ into the type $B(a)$.

Special, non-dependent, cases of the aforementioned constructors are the cartesian product $A \times B$ and the function type $A \to B$.

For any type A and $a, b : A$ elements of A, there is the Martin-Löf identity type $a =_A b$ of "(propositional) equalities" between a and b. We often omit the subscript A and hence simply write $a = b$.

One of the most salient features of univalent foundations is the univalence axiom. Intuitively, it says that any construction expressible in intensional type theory is invariant under equivalence of types. What is equivalence of types? The reader can think of it as isomorphism of types: two types A and B are isomorphic if there are maps $f : A \to B$ and $g : B \to A$ such that both composites $f \circ g$ and $g \circ f$ are pointwise equal (with respect to propositional equality) to the identity function. While the definition of equivalence is more refined than that of an isomorphism of types, it is the case that any isomorphism gives rise to an equivalence, that is, two types are isomorphic if and only if they are equivalent. The univalence axiom is stated for a particular given universe. Define, for a fixed universe $\mathcal{U}$, the canonical map

$$\mathsf{idtoeqv} : \prod_{A,B:\mathcal{U}} A = B \to A \simeq B$$

from identities to equivalences between A and B; it is defined by identity elimination, mapping the reflexivity term $\mathsf{refl}_A : A = A$ to the identity equivalence on A. The universe $\mathcal{U}$ is called **univalent** if for any A and B in $\mathcal{U}$, the map $\mathsf{idtoeqv}_{A,B}$ is an equivalence.

The univalence axiom has a number of desirable consequences—provable **inside** the theory—which can be subsumed by the term "equivalence principle": The equivalence principle says, intuitively, that reasoning about mathematical objects should be invariant under an appropriate notion of "equivalence" for those objects. In the foundation we work in, the equivalence principle can be proved for function types (function extensionality), for mathematical structures such as groups and rings [14], and for categories [7].

A second salient feature of univalent foundations is its **internal** notion of **propositions** and **sets**. A type A is called a proposition if it satisfies the (propositional) "proof irrelevance" principle, that is, if one can construct a term of type

$$\mathsf{isProp}(A) := \prod_{x,y:A} x = y \ .$$

Furthermore, a type A is called a set if all of its identity types are propositions, that is, if one can construct a term of type

$$\mathsf{isSet}(A) := \prod_{x,y:A} \mathsf{isProp}(x = y) \ .$$

These two definitions are actually special cases of a more general definition of **homotopy levels** of types. However, the general definition will not be of use in this article, and can be consulted in [29]. We call **proposition** any type that is a proposition in this sense, that is, any element of $\mathsf{Prop} := \sum_{(X:\mathcal{U})} \mathsf{isProp}(X)$, and similarly for **sets**.

2.2. **Category theory in univalent foundations.** Some category theory in univalent foundations has been developed in [7]. A category $\mathcal{C}$ is given by

- a type $\mathcal{C}_0$ of objects;

- for any $a, b : \mathcal{C}_0$, a type $\mathcal{C}(a, b)$ of morphisms from a to b;
- for any $a : \mathcal{C}_0$, an identity morphism $\mathsf{id}(a) : \mathcal{C}(a, a)$;
- for any $a, b, c : \mathcal{C}_0$, a composition function $\mathcal{C}(a, b) \to \mathcal{C}(b, c) \to \mathcal{C}(a, c)$, written $f \mapsto g \mapsto g \circ f$;
- for any $a, b : \mathcal{C}_0$ and $f : \mathcal{C}(a, b)$, we have $f \circ \mathsf{id}(a) = f$ and $\mathsf{id}(b) \circ f = f$;
- for any $a, b, c, d : A$ and $f : \mathcal{C}(a, b)$, $g : \mathcal{C}(b, c)$, $h : \mathcal{C}(c, d)$, we have $h \circ (g \circ f) = (h \circ g) \circ f$.

There is an important difference between categories as usually formalized in intensional type theory and categories as considered in [7]: in intensional type theory, categories are usually defined to come with a custom equivalence relation on the types of morphisms, which is to be read as equality relation on morphisms, specified for each category individually (see, e.g., [20], [6, Chapter 6]). This notion of category is sometimes referred to by "E-categories" [27].

In the formalization of [7], which takes place in univalent foundations, however, the authors consider morphisms of a category modulo equality as given by the identity type. That this is feasible is due to the extensional features that the univalence axiom adds to type theory, in particular, function extensionality.

The notion of category is actually more refined in [7]; two conditions must be satisfied by a category:

(i) Its hom-types $\mathcal{C}(a, b)$ need to be **sets**. This is necessary for the axioms—which talk about equality of arrows—to be propositions.

(ii) Secondly, in a category, the type of (propositional) equalities (as given by the Martin-Löf identity type) between any two objects must be equivalent to the type of isomorphisms between those objects. More precisely, to any category one defines a family of maps

$$\mathsf{idtoiso} : \prod_{a,b:\mathcal{C}_0} (a = b) \to \mathsf{iso}(a, b) \ .$$

This family of maps is defined by identity elimination, mapping $\mathsf{refl}_a : a = a$ to the identity isomorphism on a. A category $\mathcal{C}$ is called **univalent**, if for any $a, b : \mathcal{C}_0$, the map $\mathsf{idtoiso}_{a,b}$ is an equivalence.

The univalence condition for categories (ii) states, intuitively, that isomorphic objects in such a category cannot be distinguished. The equivalence principle for univalent categories, proved in [7], then says that any two equivalent such categories cannot be distinguished either, that is, the postulated invariance on objects (univalence) lifts to the categories themselves. One of the results proved below shows that our main category of interest is univalent if one starts with a univalent category (Theorem 20).

An important remark about naming: in [7], the term "precategory" is employed for categories that satisfy condition (i), and the term "category" is reserved for categories that, additionally, satisfy the univalence condition (ii). That is, the authors of [7] use the terms "precategory" and "category" for what we call "category" and "univalent category" in the present article, respectively. The rationale behind this naming convention in [7] is that the notion of categories satisfying condition (ii) should be considered to be the right notion of category, for those categories satisfy the equivalence principle. Furthermore, many important examples of categories do satisfy this condition, and the condition is closed under a lot of constructions of new categories from old categories:

- the category of sets and functions between them is univalent;
- categories of algebraic structures (groups, rings,...) are univalent;
- the functor category $[\mathcal{C}, \mathcal{D}]$ is univalent if $\mathcal{D}$ is;
- a full subcategory of a univalent category is again univalent.

More constructions of categories that preserve univalence are given below.

For the purposes of the present article, the univalence condition on categories is not essential. Indeed, no other result depends on Theorem 20 We thus choose to de-emphasize the importance of the univalence condition for categories by deviating from the naming of [7], and instead to make it explicit when considering categories that satisfy univalence.

2.3. **About UniMath.** The goal of the UniMath library is to provide a library of computer-checked mathematics formalized in (a computer implementation of) the univalent foundations. At this time, there is no computer theorem prover that implements exactly the univalent foundations as described in Section 2.1. As an approximation for such a tool, we use the COQ proof assistant [24] as a base of UniMath. However, in order to simulate working in the theory described in Section 2.1, we do not use the full language COQ provides, but restrict ourselves to the language constructors described above. In particular, there is no use of inductive types besides that of the natural numbers, and of the identity type and the type of dependent pairs, both of which are not primitives in COQ, but instead implemented via the general Inductive vernacular. Furthermore, record types are not used in UniMath; bundling of structures is instead implemented via (iterated) Sigma types.

The proof assistant COQ has recently gained a form of universe polymorphism [28]. Unfortunately, this universe management is not powerful enough for our purposes. In particular, it does not implement a form of *resizing* rule that is needed for some impredicative encodings of constructions—propositional truncation in particular, as described by Voevodsky [31, Section 4]. It was thus Voevodsky's choice to use a modified version of COQ where the checking of universe levels was deactivated, and the system hence inconsistent. In the meantime, COQ has been improved to allow the disabling of universe checking via a flag -type-in-type passed to the program, instead of modifying its source code. The UniMath library hence is based on an unmodified version of COQ, but is still working in an inconsistent system for now, while waiting for a new, more suitable universe management to be implemented.

Another difference to standard COQ is our use of the -indices-matter flag. This flag ensures that the identity type associated to a type A, lives in the same universe as the type A itself. By default, without that flag, COQ would put the identity type into the universe Prop (not to be confounded with the **homotopy level** of propositions explained in Section 2.1).

The experimental "Higher Inductive Types" (HITs), described e.g. in the HoTT book [29], are not used in UniMath.

The univalence axiom is implemented in UniMath via the Axiom vernacular of COQ. This leads to potentially non-normalizing terms, when using the axiom or any of its consequences—such as function extensionality. We do not experience any problems related to non-normalization, since we only use the univalence axiom (indirectly by using function extensionality) for proving propositions, not for specifying operations.

3. Preliminaries

Categories, functors and natural transformations are defined in [7]. Some more concepts and notation are defined in the following:

For functors $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{E}$, we write $G \cdot F : \mathcal{C} \to \mathcal{E}$ for their composition. We use the same notation for composition of a functor with a natural transformation (sometimes called "whiskering"), as in $\tau \cdot F$ and $G \cdot \tau$.

Definition 1 (pointed functors). Let $\mathcal{C}$ be a category. We denote by $\mathbf{Ptd}(\mathcal{C})$ the category of pointed endofunctors on $\mathcal{C}$, an object of which is a pair (X, η) of an endofunctor X on $\mathcal{C}$ and a natural transformation $\eta : \mathsf{Id} \to X$, called a "point" of X, where Id is the identity functor on $\mathcal{C}$. Morphisms of pointed functors are natural transformations between the underlying endofunctors that are compatible with the chosen points. Call U the forgetful functor from $\mathbf{Ptd}(\mathcal{C})$ to the underlying endofunctor category $[\mathcal{C}, \mathcal{C}]$ (in particular, for a morphism f, Uf is f, but its compatibility with the points is not taken into account in the type information—justifying to confuse Uf and f in the rest of the paper).

Definition 2 (monoidal structure on functor categories). The monoidal structure on the endofunctor category $[\mathcal{C}, \mathcal{C}]$ given by composition extends to $\mathbf{Ptd}(\mathcal{C})$. We denote by $\alpha_{X,Y,Z} : X \cdot (Y \cdot Z) \simeq (X \cdot Y) \cdot Z$, $\rho_X : \mathsf{Id} \cdot X \simeq X$ and $\lambda_X : X \cdot \mathsf{Id} \simeq X$ the monoidal isomorphisms.

Remark 3. In [23], the authors implicitly assume the monoidal structures on $[\mathcal{C}, \mathcal{C}]$ and $\mathbf{Ptd}(\mathcal{C})$ to be strict. In univalent foundations, "strict" should mean "the same modulo definitional equality"; the monoidal structures are not strict for this notion of strictness. Instead, we need to explicitly insert the isomorphisms (which correspond to propositional equalities in *univalent* categories, but that shall not be of importance in the following). Note, however, that those isomorphisms are given by families of identity morphisms, and thus do not carry any information at all; they are merely needed to formally adjust the type of source and target functors of the natural transformations involved in order to allow composing two natural transformations which would not be composable otherwise. Indeed, composability of two natural transformations $\alpha : F \to G$ and $\beta : G' \to H$ depends on G being *definitionally equal* to G'.

Definition 4 (algebras of a functor). For an endofunctor $F : \mathcal{C} \to \mathcal{C}$, the category $\mathbf{Alg}(F)$ of **algebras** has, as objects, pairs (X, α) of an object $X : \mathcal{C}_0$ and a morphism $\alpha : \mathcal{C}(FX, X)$. For a given algebra (X, α), we call X the **(algebra) carrier** of the algebra. A morphism $f : \mathbf{Alg}(F)\big((X, \alpha), (X', \alpha')\big)$ is given by a morphism $f : \mathcal{C}(X, X')$ such that $f \circ \alpha = \alpha' \circ Ff$.

Convention 5. We are using the arrow symbol "$\to$" for three different things:
  (i) morphisms $f : c \to d$ in a category, as shorthand for $f : \mathcal{C}(c, d)$ (hence in particular for natural transformations as morphisms in functor categories);
  (ii) functors $F : \mathcal{C} \to \mathcal{D}$ between categories; and
  (iii) type-theoretic functions $f : A \to B$.
Information on what the arrow denotes in each occurrence will be deducible from the context.

Definition 6 (monads). For a category $\mathcal{C}$, the category $\mathsf{Mon}(\mathcal{C})$ of **monads** has, as objects, triples (T, η, μ) of an endofunctor T of $\mathcal{C}$, and natural transformations

$\eta : \mathsf{Id} \to T$ and $\mu : T \cdot T \to T$ (using our convention on natural transformations), subject to the usual monad laws. A morphism $f : \mathsf{Mon}(\mathcal{C})\big((T, \eta, \mu), (T', \eta', \mu')\big)$ is given by a natural transformation $f : T \to T'$, subject to the usual compatibility conditions.

Notice that we follow [23] in taking monad multiplication μ as third component of a monad and not the binding operation that is more widespread in computer science literature.

Convention 7. Given $d : \mathcal{D}$ and a category $\mathcal{C}$, we call $\underline{d} : \mathcal{C} \to \mathcal{D}$ the functor that is constantly d and id_d on objects and morphisms, respectively. This notation hides the category $\mathcal{C}$, which will usually be deducible from the context. In this article, $\mathcal{C}$ will always be $\mathcal{D}$.

4. Generalized Iteration in Mendler-style and fusion law

In this section we discuss "generalized iteration in Mendler-style" and a fusion law that one can prove for this iteration scheme. Both the iteration scheme and the fusion law are used in Section 7.

Lemma 8 (Generalized iteration in Mendler-style (Theorem 2 of [13] by Bird and Paterson))**.** *Let $\mathcal{C}$ be a category, and let $F : \mathcal{C} \to \mathcal{C}$ be an endofunctor on $\mathcal{C}$. Suppose $(\mu F, \mathsf{in})$ is the initial algebra of F. Let $\mathcal{D}$ be another category, and let $\mathcal{C} : L \dashv R : \mathcal{D}$ be an adjunction. Let $X : \mathcal{D}_0$ be an object of $\mathcal{D}$, and let*

$$\Psi : \mathcal{D}(L-, X) \to \mathcal{D}(L(F-), X)$$

be a natural transformation. Then there is exactly one morphism $h : L(\mu F) \to X$ such that the following diagram commutes:

We call $\mathsf{It}^L_F(\,\Psi\,) := h$ the unique morphism thus specified.

Note that, strictly speaking, the functors occurring in the type of Ψ have to be the opposites of L and F.

The link with the work by Mendler [25] is not made in the original proof [13] of the lemma. The presentation in [13] is very much oriented towards functional programming. In their notation, the natural transformation Ψ would be typed as

$$\Psi :: \forall A.\, (LA \to X) \to (L(FA) \to X)\ .$$

The existence of the right adjoint R for L is rather a matter of technical convenience: it can be replaced by asking for the preservation of colimits of chains by F and L and the preservation of initiality by L [13, Theorem 1], but we do not pursue that alternative in our formalization.

In [23], only a specialized form of generalized iteration in Mendler-style is used that is called "generalized iteration" (again with no hint to Mendler's work—see our remarks in Section 7 on the connection). The specialization consists in taking only natural transformations Ψ of a specific form (so that Ψ disappears from the formulation, as explained in [23]). In fact, we do not need the fuller generality of

generalized iteration in Mendler-style (in Sections 7 and 8) but the formulation of the fusion law to come next is more natural in the more general setting (no fusion law was needed in [23] since no *morphisms* of heterogeneous substitution systems were considered there).

The next lemma shows a sufficient condition for two applications of the iterator $\mathsf{It}(-)$ to be related:

Lemma 9 (Fusion law). *Suppose the data as given in Lemma 8. Additionally, let $L' : \mathcal{C} \to \mathcal{D}$ be a functor, $X' : \mathcal{D}_0$ be an object of $\mathcal{D}$, let*

$$\Psi' : \mathcal{D}(L'-, X') \to \mathcal{D}(L'(F-), X')$$

be a natural transformation with type analogous to that of Ψ, and let

$$\Phi : \mathcal{D}(L-, X) \to \mathcal{D}(L'-, X')$$

be a natural transformation. Then we have

$$\Phi_{\mu F}\big(\mathsf{It}_F^L(\Psi)\big) = \mathsf{It}_F^{L'}(\Psi')$$

if

$$\Phi_{F\mu F} \circ \Psi_{\mu F} = \Psi'_{\mu F} \circ \Phi_{\mu F} \ .$$

The name "fusion law" is wide-spread in functional programming for means to eliminate the creation of some extra structure, here the subsequent calculation of $\Phi_{\mu F}$ for the result $\mathsf{It}_F^L(\Psi)$ of the iteration over μF is "fused" into one single iteration over μF—the right-hand side of the conclusion.

The version of this fusion law with X and X' the same object of $\mathcal{D}$ and instantiated to the special situation of generalized folds (see Section 7) has been found by Bird and Paterson [13] (see right before their Theorem 1). While we will only use the fusion law for generalized folds (in Section 7), it is necessary to have the liberty in choosing X and X' separately. The proof itself is a matter of verifying that the left-hand side satisfies the defining equation (embodied in the commuting diagram in Lemma 8) of the right-hand side. This also settles existence of the right-hand side, which is why we did not require a right adjoint for L', which would have allowed us to invoke Lemma 8 also for Ψ'. (In our formalization, we did not implement this subtlety but require a right adjoint for L', in order to use the definition of the $\mathsf{It}(-)$ operator underlying the formalization of Lemma 8.)

5. The category of heterogeneous substitution systems

In [23], implicitly there is a notion of signature. Here, we make this definition explicit and adapt it to the lack of strictness of our monoidal structures on endofunctors (see Definition 2) – recall that U "forgets" the points of pointed functors:

Definition 10 (Signature). Given a category $\mathcal{C}$, a **signature** is a pair (H, θ) of an endofunctor H on $[\mathcal{C}, \mathcal{C}]$ and a natural transformation $\theta : (H-) \cdot U\sim \to H(- \cdot U\sim)$ between functors $[\mathcal{C}, \mathcal{C}] \times \mathbf{Ptd}(\mathcal{C}) \to [\mathcal{C}, \mathcal{C}]$ such that

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

and

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

In practice, a signature is given by a family of *arities*, each arity specifying the type of a term constructor. The above definition of signature is modular in the sense that building a signature from arities corresponds to taking an amalgamated sum. This is explained in detail in Section 8, to which we refer for an example of signature.

Note that while the definition of signature does not require the base category $\mathcal{C}$ to have coproducts, this is a requirement for most signatures that we consider in practice, and in particular for the example of Section 8. It also is a requirement for the definition of "models" of that signature, see Definition 13.

Convention 11. From now on, we assume the category $\mathcal{C}$ to have (specified) coproducts. We denote by $\mathsf{inl}_{A,B} : A \to A + B$ and $\mathsf{inr}_{A,B} : B \to A + B$ the maps into the coproduct. We omit the subscripts of inl and inr when possible without ambiguity.

Remark 12. The notion of signature introduced in Definition 10 encompasses "polynomial" signatures like the ones described in [15] and [26]. In fact, it is strictly more general in that it also encompasses the arity of explicit flattening—the Example 33 we discuss in detail in Section 8—that is not captured by the other works mentioned above.

For a given signature (H, θ), we are interested in $(\underline{\mathsf{Id}} + H)$-algebras (T, α). For such an algebra, the natural transformation $\alpha : \mathsf{Id} + HT \to T$ decomposes into two $[\mathcal{C}, \mathcal{C}]$-morphisms $\eta : \mathsf{Id} \to T$, $\tau : HT \to T$ defined by

$$(5.1) \qquad \eta = \alpha \circ \mathsf{inl}_{\mathsf{Id}, HT} \qquad \text{and} \qquad \tau = \alpha \circ \mathsf{inr}_{\mathsf{Id}, HT} \ .$$

The pair (T, η) is an object in the category of pointed functors (see Definition 1).

Intuitively, in the case where $\mathcal{C} = \mathbf{Set}$, the transformation η corresponds to viewing variables $x : X$ as "terms", that is, as elements of TX whereas $\tau : HT \to T$ represents the recursive constructors specified by H.

Definition 13 (Def. 5 of [23], Heterogeneous substitution system of a signature)**.** We call (T, α) a *heterogeneous substitution system* (or "hss" for short) for (H, θ), if, for every $\mathbf{Ptd}(\mathcal{C})$-morphism $f : (Z, e) \to (T, \eta)$, there exists a unique $[\mathcal{C}, \mathcal{C}]$-morphism $h : T \cdot Z \to T$, denoted $\{f\}$, satisfying

$$
\begin{array}{ccc}
Z + (HT) \cdot Z & \xrightarrow{\alpha \cdot Z} & T \cdot Z \\
{\scriptstyle \mathsf{id} + \theta_{T,(Z,e)}} \downarrow & & \downarrow{\scriptstyle h} \\
Z + H(T \cdot Z) & & \\
{\scriptstyle \mathsf{id} + Hh} \downarrow & & \downarrow{\scriptstyle h} \\
Z + HT & \xrightarrow{[f,\tau]} & T
\end{array}
\qquad \text{i.e.,} \qquad
\begin{array}{ccc}
Z & \xrightarrow{\eta \cdot Z} T \cdot Z \xleftarrow{\tau \cdot Z} & (HT) \cdot Z \\
 & \searrow{\scriptstyle f} \ \downarrow{\scriptstyle h} & \downarrow{\scriptstyle \theta_{T,(Z,e)}} \\
 & T \xleftarrow{\ \tau\ } & H(T \cdot Z) \\
 & & \downarrow{\scriptstyle Hh} \\
 & & HT
\end{array}
$$

For a substitution system $(T, \alpha, \{-\})$, we call T its **carrier**, thus extending the convention of Definition 4.

Notice that the quantification is implicitly also over all pointed endofunctors (Z, e) on $\mathcal{C}$.

In the following, we sometimes omit the word "heterogeneous" when talking about heterogeneous substitution systems.

Remark 14. Being equipped with a "bracket" operation $\{-\}$ is a proposition on $(\underline{\mathsf{Id}} + H)$-algebras.

Notice that we call the operation a bracket operation although we write it with braces, to distinguish it from the bracket notation used for parallel substitution in the introduction.

The statement of the following lemma is mentioned, but not proven in [23]:

Lemma 15. *The operation $\{-\}$ is a natural transformation*

$$\mathbf{Ptd}(-, (T, \eta)) \to [\mathcal{C}, \mathcal{C}](T \cdot U-, T) \ .$$

Note that the substitution operation given by the bracket is not categorical in the sense that it is not given by a universal property. This is due to the fact that we prefer an operational point of view, where things actually compute, over a categorical one. Having substitution given as an operation rather than via a universal property is also crucial for obtaining a monad, that is, for the main theorem of [23, Thm. 10].

Definition 16 (Category of substitution systems). Given (H, θ) as before, the category $\mathsf{hss}(H, \theta)$ has, as objects, heterogeneous substitution systems as in Definition 13. A morphism of substitution systems is an algebra morphism that is compatible with the bracket $\{\}$ on either side. In terms of η and τ as defined in Equation (5.1), a morphism from $(T, \eta, \tau, \{\})$ to $(T', \eta', \tau', \{\}')$ is a natural transformation $\beta : T \to T'$ such that the following diagrams commute:

$$
\begin{array}{ccc}
\mathsf{Id} \xrightarrow{\eta} T & HT \xrightarrow{\tau} T & T \cdot Z \xrightarrow{\{f\}} T \\
\eta' \searrow \downarrow \beta & H\beta \downarrow \quad \downarrow \beta & \beta \cdot Z \downarrow \quad \downarrow \beta \\
T' & HT' \xrightarrow{\tau'} T' & T' \cdot Z \xrightarrow{\{\beta \circ f\}'} T'
\end{array}
$$

Here, the first and second diagram express the property of β being an algebra morphism, and the third diagram expresses compatibility of β with substitution on either side.

Note that the composite $\beta \circ f$ in the last diagram is the composite in the category of **pointed** endofunctors, that is, the definition of that composite uses commutativity of the first diagram.

Remark 17. Similarly to Remark 14, being compatible with the brackets on either side is a proposition on algebra morphisms.

We now study the category $\mathsf{hss}(H, \theta)$ of substitution systems associated to a signature in more detail, in particular with respect to the particular foundations we are working in. The main objective of the rest of the section is Theorem 20: the category $\mathsf{hss}(H, \theta)$ is univalent if the base category $\mathcal{C}$ is.

Remarks 14 and 17 together show that the category of $\mathsf{hss}(H, \theta)$ can be obtained as a *subcategory* of the category of $(\underline{\mathsf{Id}} + H)$-algebras in the following sense:

Definition 18. A **subcategory** of a category $\mathcal{C}$ is given by a predicate $P : \mathcal{C}_0 \to \mathsf{Prop}$ and a family of predicates $P_{a,b} : P(a) \times P(b) \times \mathcal{C}(a, b) \to \mathsf{Prop}$ that is closed under identity and composition in the sense that

- for any $a : \mathcal{C}_0$ satisfying P, have a proof of $P_{a,a}(\mathsf{id}(a))$ and
- for any $a, b, c : \mathcal{C}_0$ satisfying P, and for any $f : \mathcal{C}(a, b)$ and $g : \mathcal{C}(b, c)$, have a map $P_{a,b}(f) \to P_{b,c}(g) \to P_{a,c}(g \circ f)$.

We suppress the arguments of type $P(a)$ and $P(b)$ when discussing the predicate $P_{a,b}(f)$, since those arguments are unique.

A subcategory of $\mathcal{C}$ is—better, gives rise to—a category $\mathcal{C}_P$; objects are of the form $\sum_{(x:\mathcal{C}_0)} P(x)$, and morphisms $(f, p_f) : \mathcal{C}_P\big((a, p_a), (b, p_b)\big)$ are pairs of a morphism $f : \mathcal{C}(a, b)$ of $\mathcal{C}$ together with a proof $p : P_{a,b}(f)$.

Given a signature (H, θ), define a subcategory of the category of $(\underline{\mathsf{Id}}+H)$-algebras via the predicates of Remarks 14 and 17. The resulting category is clearly isomorphic to $\mathsf{hss}(H, \theta)$ in the sense of [7, Definition 6.9].

Note that isomorphic categories are equal modulo propositional equality [7, Definition 6.16], and hence share all properties definable in type theory. We thus give up the distinction between the category $\mathsf{hss}(H, \theta)$ and the subcategory of $(\underline{\mathsf{Id}} + H)$-algebras it is isomorphic to.

A subcategory is called **replete**, when it is closed under isomorphism, that is, when, for $f : \mathsf{iso}_{\mathcal{C}}(a, b)$ and $P(a)$, it follows that $P(b)$ and $P_{a,b}(f)$.

Lemma 19. *The category $\mathsf{hss}(H, \theta)$ is a replete subcategory of the category of $(\underline{\mathsf{Id}} + H)$-algebras.*

Proof. Given a substitution system $(T, \alpha, \{-\})$, an algebra (T', α') and an algebra isomorphism $\beta : (T, \alpha) \to (T', \alpha')$, we define a bracket $\{-\}'$ on (T', α') as follows: for a given pointed morphism $f : (Z, e) \to (T', \eta')$, we define $\{f\}'$ as the composition

$$\{f\}' := \beta \circ \{\beta^{-1} \circ f\} \circ \beta^{-1} \cdot Z \;:\; T' \leftarrow T \leftarrow T \cdot Z \leftarrow T' \cdot Z$$

The morphism $\{f\}'$ thus defined satisfies the equations of Definition 13,

$$f = \{f\}' \circ \eta' \cdot Z$$
$$\{f\}' \circ \tau' \cdot Z = \tau' \circ H(\{f\}') \circ \theta_{T',(Z,e)} \;;$$

the calculation is routine. Concerning the uniqueness of $\{f\}'$, suppose h such that these equations with h in place of $\{f\}'$ are satisfied. We have to show that $h = \beta \circ \{\beta^{-1} \circ f\} \circ \beta^{-1} \cdot Z$. Equivalently, one can show that

(5.2) $$\{\beta^{-1} \circ f\} = \beta^{-1} \circ h \circ \beta \cdot Z \;,$$

which follows from the uniqueness of $\{-\}$: it suffices to show that the right-hand side of (5.2) satisfies the equations involving η and τ. We thus have equipped (T', α') with a (necessarily unique) substitution operation.

The fact that β is compatible with $\{-\}$ and $\{-\}'$, and hence in the subcategory, is a routine calculation. $\square$

Theorem 20. *The category $\mathsf{hss}(H, \theta)$ is univalent if $\mathcal{C}$ is.*

Proof. Combine Lemmas 21, 23, 22 and 19. More precisely, if $\mathcal{C}$ is univalent, so is $[\mathcal{C}, \mathcal{C}]$, and thus also the category of $(\underline{\mathsf{Id}}+H)$-algebras on $[\mathcal{C}, \mathcal{C}]$. Finally, the category $\mathsf{hss}(H, \theta)$ is univalent as a replete subcategory of that of $(\underline{\mathsf{Id}} + H)$-algebras. $\square$

The following lemmas state closure properties of the property of being univalent:

Lemma 21. *The category of algebras of a functor $F : \mathcal{C} \to \mathcal{C}$ is univalent if $\mathcal{C}$ is.*

Proof. This lemma is proved in the file `CategoryTheory/FunctorAlgebras.v` of the `UniMath` library. $\square$

The next lemma is originally due to Hofmann and Streicher [19]; and is also proved in Thm. 4.5 of [7]:

Lemma 22. *The category of functors* $[\mathcal{C}, \mathcal{D}]$ *is univalent if the target category* $\mathcal{D}$ *is.*

The category of hss contains all the isomorphisms of the category of $(\underline{\mathsf{Id}} + H)$-algebras, for which source and target are substitution systems. This is sufficient to inherit univalence from the category of algebras:

Lemma 23. *Let* $\mathcal{C}$ *be a univalent category and let* $P : \mathcal{C}_0 \to \mathsf{Prop}$ *and* $P_{a,b} :$ $\mathcal{C}(a,b) \to \mathsf{Prop}$ *define a subcategory* $\mathcal{C}_P$ *of* $\mathcal{C}$*. Then* $\mathcal{C}_P$ *is univalent if, for any objects* (a, p_a) *and* (b, p_b) *of* $\mathcal{C}_P$*, and for any* iso*morphism* $f : \mathsf{iso}_{\mathcal{C}}(a, b)$ *from* a *to* b*, we have* $P_{a,b}(f)$*.*

In particular, replete subcategories of univalent categories are univalent.

Proof. For (a, p_a) and (b, p_b) objects of $\mathcal{C}_P$, we have

$$(a, p_a) =_{\mathcal{C}_P} (b, p_b) \ \simeq \ a =_{\mathcal{C}} b \ \simeq \ \mathsf{iso}_{\mathcal{C}}(a, b) \ \simeq \ \mathsf{iso}_{\mathcal{C}_P}((a, p_a), (b, p_b))$$

and this equivalence, from left to right, is equal to idtoiso. $\qquad\square$

This concludes our study of the category of substitution systems associated to a signature.

6. FROM SUBSTITUTION SYSTEMS TO MONADS

One of the most important results of Matthes and Uustalu's work [23] is the construction of a monad from any substitution system:

Theorem 24 ([23], Thm. 10)**.** *If an* $(\underline{\mathsf{Id}} + H)$-*algebra* (T, α) *forms a heterogeneous substitution system for* (H, θ) *for some* θ*, then* $(T, \eta, \{\mathsf{id}_{(T,\eta)}\})$ *is a monad.*

See Section 9 for some comments on technical challenges we had to overcome for the formalization of its proof.

It is natural to ask whether this map extends to *morphisms*, and indeed it does:

Theorem 25. *The map from heterogeneous substitution systems to monads defined in* [23, Thm. 10] *is the object map of a functor* $\mathsf{hss}(H, \theta) \to \mathsf{Mon}(\mathcal{C})$*.*

Proof. Given any morphism $\beta : (T, \eta, \tau, \{\}) \to (T', \eta', \tau', \{\}')$ of hss, the underlying natural transformation $\beta : T \to T'$ needs to be proven compatible with the multiplications $\mu^T := \{\mathsf{id}_{(T,\eta)}\}$ and $\mu^{T'}$ of the monadic structures on T and T' defined in [23, Thm. 10]. This is an easy consequence of the compatibility of β with $\{\}$ and $\{\}'$. $\qquad\square$

The functor from substitution systems to monads is faithful, but not full. Intuitively, the lack of fullness stems from the fact that the axioms of a monad morphism do not specify compatibility of the mapping with the "inner nodes" of an expression, but only at the leaves, that is, in the case of a variable.

Lemma 26. *The functor of Definition 25 is faithful.*

Proof. Two parallel monad morphisms are equal if their underlying natural transformations are, and the analogous statement is true for morphisms of substitution systems. $\qquad\square$

Remark 27. The functor of Definition 25 is not full. For instance, choose $\mathcal{C} = \textbf{Set}$, and take a signature with two copies $\mathtt{app}$ and $\mathtt{app'}$ (of the same arity) of an "application" constructor, see Definition 31 in Section 8. Take the initial substitution system associated to that signature (as constructed via Theorems 28 and 29 in Section 7), and define an endomorphism on it that maps $\mathtt{app}$ to $\mathtt{app'}$ recursively, and is the identity on the other constructors. This yields a monad morphism, but not a morphism of substitution systems; indeed, the second diagram of Def. 16 does not commute—any endomorphism on that substitution system must be the identity morphism.

7. Lifting initiality through a fusion law

The starting point of this section is a result from [23], which gives one way to define substitution systems and which comes from a very specific instance of Lemma 8. As a first instantiation step, take in that lemma $[\mathcal{C}, \mathcal{C}]$ for $\mathcal{C}$ and $\mathcal{D}$ and the *reduction functor* $- \cdot Z$ for L, for any endofunctor Z of $\mathcal{C}$. This is the general situation of the "gfolds" of Bird and Paterson [13], and (the carriers of) the corresponding initial F-algebras are called "nested datatypes" [11]. As Bird and Paterson recall, the assumption of having a right adjoint to the reduction functor means that right Kan extensions along those Z exist. In the context of functional programming with impredicative polymorphism, these right Kan extensions even exist in a computational way (although the full categorical properties of Kan extensions are not reflected computationally) [4]. We will not further develop the categorical semantics of those programming languages. The previous remarks should make it plausible that the following theorem rests on "reasonable" technical conditions. If program verification is aimed at in an intensional setting, replacements for the categorical notions have to be found, and yet different schemes of generalized iteration have to be studied in order to combine expressivity, termination guarantees and program verification in the same framework [22] (using Coq very differently from the $\mathtt{UniMath}$ approach).

Theorem 28 ([23], Thm. 15). *Let (H, θ) be a signature. If $[\mathcal{C}, \mathcal{C}]$ has an initial $(\underline{\mathsf{Id}} + H)$-algebra and a right adjoint for the functor $- \cdot Z : [\mathcal{C}, \mathcal{C}] \to [\mathcal{C}, \mathcal{C}]$ exists for every $\textbf{Ptd}(\mathcal{C})$-object (Z, e), then (T, α) defined by*

$$(T, \alpha) = (\mu(\underline{\mathsf{Id}} + H), \mathsf{in}_{\underline{\mathsf{Id}}+H})$$

is a heterogeneous substitution system for (H, θ).

The proof of this theorem is by identifying, for a given $f : (Z, e) \to (T, \eta)$, the morphism $\{f\}$ as an instance of Lemma 8, both for the existence and uniqueness property. The obvious part of the instantiation is the choice of parameters mentioned above, and by setting $F := \underline{\mathsf{Id}} + H$. The essential ingredient for getting a morphism $\{f\}$ of type $\mu F \cdot Z \to T$ (here, T is even μF) is a natural transformation Ψ_f whose typing could sloppily be written as

$$\Psi_f :: \forall X : [\mathcal{C}, \mathcal{C}]. (X \cdot Z \to T) \to (FX \cdot Z \to T) \ .$$

The type of Ψ_f suggests the following problem-solving method: The original problem is that of finding a morphism of type $\mu F \cdot Z \to T$. We abstract away from μF and replace it by an arbitrary endofunctor $X : [\mathcal{C}, \mathcal{C}]$. For this arbitrary X, we have to extend a purported solution for parameter X, hence of type $X \cdot Z \to T$, to a solution for parameter FX, hence of type $FX \cdot Z \to T$. Of course, this has

to be done naturally in X, as required in Lemma 8. So, passing naturally from X to FX as parameter, the lemma even yields a (unique) solution for the least fixed-point of F as parameter. The continuity properties behind this method already for (co-)inductive types have been deeply explored by Abel [2] and extended to nested dataypes later [3].

This is the essence of schemes in Mendler's style [25]: passing from a solution in parameter X to a solution in parameter FX *uniformly* (in Mendler's original work, this was plainly universal quantification over a type variable X, in the categorical setting, this is achieved by naturality), one is guaranteed a solution in parameter μF. Lemma 8 is an instance of that idea, hence the name generalized iteration in Mendler-style.

Mendler-style gives great liberty: were are free in choosing Ψ_f of the required type (implicitly asking for naturality), but there is little guidance in finding the right one for our purpose. Guidance would, e. g., come from asking for an algebra structure on the target endomorphism T. Therefore, we instantiate the lemma further to obtain what is called "a special case of generalized iteration" by Matthes and Uustalu [23].[1] It consists in requiring an endofunctor F' on $[\mathcal{C}, \mathcal{C}]$, a natural transformation $\theta' : (F-) \cdot Z \to F'(- \cdot Z)$ and an F'-algebra $\varphi : F'T \to T$ on T, and in putting them together to obtain

$$\Psi_f(X)(h : X \cdot Z \to T) := \varphi \circ F'h \circ \theta'_X : FX \cdot Z \to T \ .$$

Its use in our present situation is then with $F' := \underline{Z} + H$, $\theta'_X := \mathsf{id} + \theta_{X,(Z,e)}$ and $\varphi := [f, \tau]$, using the datum θ of the signature and the H-algebra τ that is generically derived from α (see before Definition 13).

We remark that all of this is not optimal from a progammer's point of view (the question is then not only of soundness but of efficiency of the traversals through the data structures) and that there is the more refined notion of "generalized Mendler iteration" [4] (called GMIt^ω) as an efficient way out. The crucial idea is to generalize the problem further than finding a solution of $X \cdot Z \to T$ for parameter $X = \mu F$. An $h : X \cdot Z \to T$ consists of morphisms $h_A : X(ZA) \to TA$ for every $A : \mathcal{C}_0$, and generalized Mendler iteration asks even for operations $h_f : XB \to TA$ for any $B : \mathcal{C}_0$ and $f : B \to ZA$. Taking for f the identity morphism on ZA, one gets the desired components of the solution in the end. The gain in efficiency comes from the combination of a fold and a map in this scheme—enforced just by these types in the polymorphic formulation of [4].

Also for generalized Mendler iteration, there is a formulation in more conventional terms of algebras, called "generalized refined conventional iteration" [4], which captures in particular the efficient folds of Martin, Gibbons and Bayley [21]. For generalized Mendler iteration, there is also a means of verification in usual intensional Coq, using category theory only as a motivation and not as the mathematical framework [22].

We augment the previous theorem by showing that the constructed substitution system is initial:

Theorem 29. *The substitution system* $(T, \alpha, \{\})$ *constructed in Lemma 28 is initial in* $\mathsf{hss}(H, \theta)$.

---

[1]The instantiation with $- \cdot Z$ for L can also be formulated in a less homogeneous setting where not only endofunctor categories intervene [23, Section 2.3].

In order to prove Theorem 29, it suffices to show that, for any given substitution system $(T', \alpha', \{\}')$, the initial morphism **of algebras**

$$! : (T, \alpha) \to (T', \alpha')$$

is compatible with the operations $\{\}$ (defined in the proof of Lemma 28) and $\{\}'$. That is, we need to show that, for any $f : (Z, e) \to (T, \eta)$,

$$(7.1) \qquad\qquad ! \circ \{f\} = \{! \circ f\}' \circ (! \cdot Z) \ .$$

Using the fusion law (Lemma 9), we show that both sides of (7.1) are equal to the application of an iterator. More precisely, we use the fusion law for the left-hand side, knowing the explicit definition of $\{f\}$ as an iterator, described above, to establish equality with $\mathsf{It}_F^{-\cdot Z}(\Psi_f)$, where we define

$$\Psi_f(X)(h : X \cdot Z \to T') := [! \circ f, \tau' \circ Hh \circ \theta_{X,(Z,e)}] : FX \cdot Z \to T' \ .$$

Once the premises of the fusion law established, we can show equality with the right-hand side of (7.1) by verifying that the defining equations of $\mathsf{It}_F^{-\cdot Z}(\Psi_f)$ are fulfilled by the right-hand side.

8. A WORKED EXAMPLE: FLATTENING OF EXPLICIT SUBSTITUTION

In practice, a signature is often a family of arities, each arity specifying the type of one term constructor. A typical example is a typeful version of de Bruijn indices for pure (untyped) λ-calculus, where, intuitively, the equation

$$TA = A + TA \times TA + T(1 + A)$$

has to be solved, giving in TA the set of λ-terms having free variables among A (cf. the introduction), where the last summand represents λ-abstraction that abstracts the variable corresponding to the extra element of $1 + A$. This example is developed in [23] but originates in [9, 12].

In our formalism (that of [23]), we do not need to distinguish between arities and signatures. Intuitively, an arity is a signature that is not obtained as a proper sum of two other signatures. In particular, a single arity constitutes a signature, and we can "glue" signatures together to obtain a new signature:

Lemma 30 (Sum of signatures). *Let (H, θ) and (H', θ') be two signatures. Then $(H + H', \theta + \theta')$ is a signature.*

This lemma is important for our main example: indeed, we consider two signatures, where one is obtained from the other by extending the language (better: its signature) by one additional term constructor (better: arity).

To this end, we need the base category $\mathcal{C}$ to come equipped with some extra structure: for the remainder of this section, we assume $\mathcal{C}$ to have (specified) products, coproducts and a terminal object. An example of such a category is the (univalent) category **Set** of sets (see Section 2), which has all limits and colimits.

We continue the case study in [23] on λ-calculus without and with a form of explicit substitution—"explicit flattening". In order to do so, we first present the functors H and natural transformations θ corresponding to the arities of application, abstraction, and explicit flattening, respectively:

Definition 31 (application). The signature of application is given by pointwise product, inherited from the base category $\mathcal{C}$:

$$H^{\mathsf{App}}(T) := T \times T \ .$$

The natural transformation θ^{App} is given pointwise by the identity,

$$\theta^{\mathsf{App}}_{X,(Z,e)} : (X \times X) \cdot Z \to (X \cdot Z) \times (X \cdot Z) \ .$$

The fact that the identity suffices here corresponds to the triviality of first-order operations in substitution (which is plainly homomorphic on those operations).

Definition 32 (abstraction)**.** Abstraction in our context is defined by precomposition with a coproduct, corresponding to "context extension":

$$H^{\mathsf{Abs}}(T) := T \cdot \mathsf{option} \ ,$$

where $\mathsf{option}(X) := 1 + X$ represents the context X extended by one distinguished element $\mathsf{inl}_{1,X}(\star)$. The "strength" θ is defined as

$$\theta^{\mathsf{Abs}}_{X,(Z,e)}(A) := X[\, e_{1+A} \circ \mathsf{inl}_{1,A}, Z\mathsf{inr}_{1,A} \,] : X(1 + ZA) \to X(Z(1 + A)) \ .$$

The defined strength embodies the usual lifting needed for substitution in de Bruijn representations of λ-abstraction.

Definition 33 (explicit flattening)**.** The flattening signature is defined by selfcomposition,

$$H^{\mathsf{Flatten}}(T) := T \cdot T \ ,$$

and the corresponding strength requires the unit e of the pointed endofunctor (Z, e) to be inserted in the right place:

$$\theta^{\mathsf{Flatten}}_{X,(Z,e)} := X \cdot e \cdot X \cdot Z : X \cdot X \cdot Z \to X \cdot Z \cdot X \cdot Z \ .$$

Note that the flattening signature cannot be dealt with in a framework with a fixed enumeration of variable names and shows, already on the syntactic side, the most simple case of "true nesting" in nested datatypes (see, e.g., [4]). Notice that the highly parameterized type already suggests the right definition. For its mainly used instance $\theta^{\mathsf{Flatten}}_{T,(T,\eta)}$, with T and η components of the obtained substitution system, its type $T^3 \to T^4$ hardly suggests a canonical definition.

These signatures are now combined, as per Lemma 30, to obtain the signatures we are mainly interested in:

Definition 34 (λ-calculus)**.** The signature Λ is obtained as the sum of the signatures of Defs. 31 and 32.

Definition 35 (λ-calculus with explicit flattening)**.** The signature Λ^μ is obtained as the sum of the signatures of Defs. 34 and 33.

For the purpose of this example, we assume the signatures Λ and Λ^μ to have initial substitution systems. By Lemma 28 we get those if we assume that their underlying initial algebras exist. (For a remark on the construction of initial algebras, see Section 10.) We denote the initial substitution systems by $(\mathsf{Lam}, \alpha, \{\})$ and $(\mathsf{Lam}^\mu, \alpha^\mu, \{\}^\mu)$, respectively. Intuitively, they solve the equation in T given in the first paragraph of this section, and the following in T', respectively:

$$T'A = A + T'A \times T'A + T'(\mathsf{option}\, A) + T'(T'A) \ .$$

Why is Lam^μ supposed to represent λ-calculus with explicit flattening? Coming back to parallel substitution on T ($= \mathsf{Lam}$), as mentioned in the introduction, we may study the substitution rule $f := \lambda x^{TB}.x$ of type $TB \to TB$. Then, $\mu_B := [f] : T(TB) \to TB$ can be interpreted as doing the following: in a term whose free

variables have as names terms over B, those names are replaced by themselves, but now seen as terms that are "integrated" into the result term. In other words, μ_B removes the "cross section" between the trunk of the term and the term-like variable leaves. Invoking Theorem 24 for $(\mathsf{Lam}, \alpha, \{\})$, one obtains $\mu := \{\mathsf{id}_{(\mathsf{Lam}, \eta)}\} :$ $\mathsf{Lam} \cdot \mathsf{Lam} \to \mathsf{Lam}$ as monad multiplication on the monad of λ-terms, and the above-mentioned parallel substitution can then be derived generically, so as to obtain its components μ_B with the described behaviour. In other words, the generic notion of monad multiplication appears to have the behaviour of "flattening" a nested term structure of type $T(TB)$ into one of type B (for every B). Now, Lam^μ even has a term constructor, corresponding to the injection of the last summand of the above equation into the left-hand side, and so, the constructor is of type $\mathsf{Lam}^\mu \cdot \mathsf{Lam}^\mu \to \mathsf{Lam}^\mu$, which is of the same type as the monad multiplication that is obtained by invoking Theorem 24 for $(\mathsf{Lam}^\mu, \alpha^\mu, \{\}^\mu)$. As a constructor, this operation does *not* denote the result of the flattening (here, even for the extended syntax), but is a formal syntactic element and is thus termed an "explicit flattening". Already in [23], it was shown that those explicit flattenings can be resolved by evaluating any term with explicit flattenings (from $\mathsf{Lam}^\mu A$ for some A) into a term without explicit flattenings (in $\mathsf{Lam}A$). We continue this case study by using our extra categorical structure on substitution systems.

In the following, our goal is to construct a morphism of substitution systems from Lam^μ to Lam. This is not quite precise and needs refinement, since a priori, those two substitution systems are not in the same category. More precisely, we are going to build a substitution system for the signature Λ^μ, the underlying carrier of which is the carrier Lam. To this end, we need to construct two ingredients: firstly, we need a natural transformation $\mu^{\mathsf{Lam}} : H^{\mathsf{Flatten}}(\mathsf{Lam}) \to \mathsf{Lam}$ in order to obtain a structure of $\underline{\mathsf{Id}} + \Lambda^\mu$-algebra on Lam. Secondly, we equip this $\underline{\mathsf{Id}} + \Lambda^\mu$-algebra with a bracket operation—which, of course, must be shown compatible with the $\underline{\mathsf{Id}} + \Lambda^\mu$-algebra structure in the sense of the diagram of Definition 13.

Once this is done, we obtain, by initiality, a morphism of hss from the initial hss of Λ^μ to the newly constructed one, the underlying algebra morphism of which is a morphism from Lam^μ to Lam that "does the right thing": mapping explicit substitution to substitution.

Definition 36 (representation of flattening on Lam)**.** Let $\mu^{\mathsf{Lam}} : H^{\mathsf{Flatten}}(\mathsf{Lam}) \to \mathsf{Lam}$ be given by

$$\mu^{\mathsf{Lam}} := \{\mathsf{id}_{\mathsf{Lam}}\} : \mathsf{Lam} \cdot \mathsf{Lam} \to \mathsf{Lam} \ .$$

Lemma 37 (substitution system of Λ^μ on Lam)**.** *The pair* $(\mathsf{Lam}, [\,\alpha, \mu^{\mathsf{Lam}}\,])$ *is an* $\underline{\mathsf{Id}} + \Lambda^\mu$*-algebra. (Here, we have implicitly used associativity of the coproduct.)*

We define a bracket operation $\{\}^{\mathsf{Flatten}}$ *on this algebra by setting, for* (Z, e) *and* $f : (Z, e) \to (\mathsf{Lam}, \eta)$,

$$\{f\}^{\mathsf{Flatten}} := \{f\} \ .$$

This assignment yields a bracket operation on that algebra, and hence a substitution system $(\mathsf{Lam}, [\,\alpha, \mu^{\mathsf{Lam}}\,], \{\}^{\mathsf{Flatten}})$ *for the signature* Λ^μ*.*

Proof. We need to show that $\{-\}^{\mathsf{Flatten}}$ satisfies the equations of a bracket operation, see Definition 13. The diagrams can be checked for any "arity" individually, and for η, App and Abs, the equations to check are exactly those satisfied by Lam as a substitution system for the signature Λ. The only non-trivial equation to check

states that $\{-\}^{\mathsf{Flatten}}$ is compatible with μ^{Lam}; we have to check that

$$\{f\}^{\mathsf{Flatten}} \circ \mu^{\mathsf{Lam}} \cdot Z = \mu^{\mathsf{Lam}} \circ \mathsf{Lam}(\{f\}^{\mathsf{Flatten}}) \circ \{f\}^{\mathsf{Flatten}} \cdot \mathsf{Lam} \cdot Z \circ \mathsf{Lam} \cdot e \cdot \mathsf{Lam} \cdot Z$$

We omit the details of this calculation here, and refer instead to the formal proof.
□

We thus have two objects in the category $\mathsf{hss}(\Lambda^\mu)$, an initial object with underlying carrier Lam^μ, and the object constructed in Lemma 37, with underlying carrier Lam. By initiality, we obtain a unique morphism of hss in this category.

Definition 38. We call $\mathsf{eval} : \mathsf{Lam}^\mu \to \mathsf{Lam}$ the morphism of substitution systems obtained by initiality. This map sends application and abstraction to themselves, respectively, and it sends the explicit flattening operator to its "evaluation", that is, to a "flattened" term.

This morphism of hss gives rise, via functoriality of the monad construction (Theorem 25), to a monad morphism; it is this morphism that is studied in Example 16 of [23]. Here, we have shown how that monad morphism arises from a morphism of substitution systems.

9. About the formalization

Most of the results presented in this article have been formalized, based on the `UniMath` library [1]. More precisely, all results except for Theorem 20 and Lemmas 23 and 19 are proved in our formalization.

Our formalization started out as an independent repository, but has since been integrated into `UniMath`, as a package (subdirectory) called `SubstitutionSystems`. The formalization can be inspected by cloning the `UniMath` repository on Github, `https://github.com/UniMath/UniMath`, following the installation procedure described there.

The `UniMath` library being under active development, the organization of the packages is going to change: some code will be moved to other, more fundamental, packages. For the purpose of inspection of the package `SubstitutionSystems` as described here, it is hence convenient to stick with a particular commit of the git repository, e.g., `commit 1ead81a`. The sections of this article roughly correspond to files in the formalization:

> `GenMendlerIteration.v:` corresponds to Section 4;
> `SubstitutionSystems.v:` corresponds to Section 5;
> `MonadsFromSubstitutionSystems:` corresponds to Section 6;
> `LiftingInitial.v:` corresponds to Section 7.

The code corresponding to Section 8 is spread over several files:

> `SumOfSignatures.v:` corresponds to Lemma 30;
> `LamSignature.v:` corresponds to Definitions 31, 32, 33;
> `Lam.v:` corresponds to the rest of Section 8.

To account for the evolution that is going to happen in the `UniMath` library, we provide an "interface" file
`UniMath/SubstitutionSystems/SubstitutionSystems_Summary.v`
containing pointers to the most important formalized theorems.

TABLE 1. Lines of code of the library `SubstitutionSystems`

spec	proof	comments	
32	59	10	AdjunctionHomTypesWeq.v
90	165	102	Auxiliary.v
28	14	8	EndofunctorsMonoidal.v
70	124	27	FunctorsPointwiseCoproduct.v
70	113	7	FunctorsPointwiseProduct.v
91	116	30	GenMendlerIteration.v
28	21	7	HorizontalComposition.v
79	407	72	LamSignature.v
106	249	57	Lam.v
236	518	61	LiftingInitial.v
123	423	76	MonadsFromSubstitutionSystems.v
26	0	12	Notation.v
15	4	9	PointedFunctorsComposition.v
36	61	11	PointedFunctors.v
42	81	11	ProductPrecategory.v
22	0	10	RightKanExtension.v
82	211	40	Signatures.v
155	326	53	SubstitutionSystems.v
69	170	13	SumOfSignatures.v
1400	3062	616	total

9.1. **Statistics.** Our library consists of a bit more than 4400 loc, plus 600 lines of comments[2]. Details are given in Table 1—numbers are taken from `commit 1ead81a`. For comparison, for the same commit, the whole of `UniMath`, including our library, consists of about 37000 lines of code:

spec	proof	comments	
15053	22389	3987	total

9.2. **About performance: transparency vs. opacity.** One important aspect of computer proof assistants that are based on type theory is **computation**. Computation enables us to obtain some equalities for free. For instance, in our formalization of (co)products in a functor category $[\mathcal{C}, \mathcal{D}]$ from (co)products in the target category $\mathcal{D}$, the (co)product of two functors F and G **computes** pointwise to the (co)product of the images, that is, for instance $(F \oplus_{[\mathcal{C},\mathcal{D}]} G)(c) \equiv Fc \oplus_{\mathcal{D}} Gc$. Here, the notation $\equiv$ denotes definitional equality a.k.a. computation. This is only true for a specific construction of (co)products in functor categories, of course; in general, one can only expect $(F \oplus_{[\mathcal{C},\mathcal{D}]} G)(c) \simeq_{\mathcal{D}} Fc \oplus_{\mathcal{D}} Gc$. However, in order to keep the complexity of our proofs manageable for us, having definitional equality instead of isomorphism was crucial. We hence had to keep many category-theoretic constructions, such as (co)products in functor categories, **transparent**. Technically, this amounts to closing a proof using `Defined.` instead of `Qed.` in the COQ proof assistant.

---

[2]Note that the organization of the files is going to change over time, due to reorganization of the library. In particular, contents may get moved to other parts of `UniMath` in the future.

This lack of opacification, however, results in terms getting very large, making type checking more costly for the machine. The transparency vs. opacity issue can hence be restated as an issue of human vs. machine friendliness.

Our approach to this issue was to opacify all the terms that we could afford opacifying, either by moving them into lemmas by themselves, closing with `Qed.`, or by enclosing the corresponding sequence of tactics producing that term into an `abstract (...)` block. The inconvenience of the latter method is that the block enclosed by `abstract` must be **one** tactic (composed using the chaining semicolon), not a sequence of tactics. This method is hence only feasible for small subproofs.

Our library is quite slow to compile, due to the rather large proof terms arising when working with rank 2 functors: some `Qed.` take very long to check. A significant speedup was obtained in the file `MonadsFromSubstitutionSystems.v` by setting the option `Unset Kernel Term Sharing.`, the workings of which are unknown to us. However, this option proved useless or even increased compile time in other files, and is hence only used in that one file. It is unclear to us why this option is beneficial in that file and only there, and whether there is a guiding principle saying when this option is useful.

In our library, there is a slight duplication of code: the `UniMath` library contains a proof that colimits lift to functor categories from the target category, formalized by Ahrens and Mörtberg [8]. This result could in principle be applied to lift coproducts and products, both of which are formalized as specific colimits. However, it turned out that this approach made typechecking unfeasibly slow: indeed, the first files making use of coproducts in functor categories would stop compiling when that construction of coproducts in functor categories was plugged in. Instead, we provide a manual lifting of (co)products into functor categories in the files `FunctorsPointwiseProduct.v` and `FunctorsPointwiseCoproduct.v`, with which typechecking is reasonably fast. The latter construction applies similar principles of opacification as the general lifting of colimits; it is hence unclear to us why the latter does perform so much better than the former. We hope to clarify this issue in future work [8].

10. Conclusions

We presented, in a univalent foundation, some new results about the heterogeneous substitution systems introduced by Matthes and Uustalu [23], and showed how to obtain initial substitution systems (such as lambda calculi) from initial algebras using generalized iteration in Mendler-style.

We have not studied the construction of initial algebras in univalent foundations; this is the subject of a forthcoming work by Ahrens and Mörtberg [8].

Thanks to Paige North for discussion of the subject matter, and to Anders Mörtberg for providing feedback to a draft of this article. Thanks to the rest of the `UniMath` team, for providing a sound base for formalization, and, specifically, to Dan Grayson and Anders Mörtberg for helping maintain the code described in this article.

References

[1] UniMath. `http://unimath.org`. 3, 4, 20

[2] Andreas Abel. Termination checking with types. *ITA*, 38(4):277–319, 2004. 16

[3] Andreas Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types.* Doktorarbeit (PhD thesis), LMU München, 2006. 16

[4] Andreas Abel, Ralph Matthes, and Tarmo Uustalu. Iteration and coiteration schemes for higher-order and nested datatypes. *Theoretical Computer Science*, 333(1–2):3–66, 2005. 15, 16, 18

[5] Benedikt Ahrens. Extended Initiality for Typed Abstract Syntax. *Logical Methods in Computer Science*, 8(2):1 – 35, 2012. 3

[6] Benedikt Ahrens. *Initiality for typed syntax and semantics*. PhD thesis, Université Nice Sophia Antipolis, France, 2012. 6

[7] Benedikt Ahrens, Krzysztof Kapulkin, and Michael Shulman. Univalent categories and the Rezk completion. *Math. Struct. in Comp. Science*, 25:1010–1039, 2015. Also arXiv:1303.0584. 3, 5, 6, 7, 8, 13

[8] Benedikt Ahrens and Anders Mörtberg. Binding syntax in univalent foundations. Work in progress. 3, 22

[9] Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In Jörg Flum and Mario Rodríguez-Artalejo, editors, *Computer Science Logic, 13th International Workshop, CSL '99, Proceedings*, volume 1683 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 1999. 17

[10] Françoise Bellegarde and James Hook. Substitution: A formal methods case study using monads and transformations. *Science of Computer Programming*, 23:287–311, 1994. 2

[11] Richard Bird and Lambert Meertens. Nested datatypes. In Johan Jeuring, editor, *Mathematics of Program Construction, MPC'98, Proceedings*, volume 1422 of *Lecture Notes in Copmputer Science*, pages 52–67. Springer, 1998. 15

[12] Richard S. Bird and Ross Paterson. De Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, 1999. 17

[13] Richard S. Bird and Ross Paterson. Generalised folds for nested datatypes. *Formal Asp. Comput.*, 11(2):200–222, 1999. 3, 9, 10, 15

[14] Thierry Coquand and Nils Anders Danielsson. Isomorphism is equality. *Indagationes Mathematicae*, 24(4):1105 – 1120, 2013. In memory of N.G. (Dick) de Bruijn (1918–2012). 5

[15] Marcelo Fiore, Gordon Plotkin, and Daniele Turi. Abstract syntax and variable binding. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, LICS '99, pages 193–202, 1999. 3, 11

[16] 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. 3

[17] André Hirschowitz and Marco Maggesi. Modules over monads and initial semantics. *Inf. Comput.*, 208(5):545–564, 2010. 3

[18] André Hirschowitz and Marco Maggesi. Initial semantics for strengthened signatures. In Dale Miller and Zoltán Ésik, editors, *Proceedings 8th Workshop on Fixed Points in Computer Science, FICS 2012, Tallinn, Estonia, 24th March 2012.*, volume 77 of *EPTCS*, pages 31–38, 2012. 3

[19] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In *Twenty-five years of constructive type theory (Venice, 1995)*, volume 36 of *Oxford Logic Guides*, pages 83–111. Oxford Univ. Press, New York, 1998. 13

[20] Gérard Huet and Amokrane Saïbi. Constructive Category Theory. In *In Proceedings of the Joint CLICS-TYPES Workshop on Categories and Type Theory, Goteborg*. MIT Press, 1998. 6

[21] Clare Martin, Jeremy Gibbons, and Ian Bayley. Disciplined, efficient, generalised folds for nested datatypes. *Formal Aspects of Computing*, 16(1):19–35, 2004. 16

[22] Ralph Matthes. Map fusion for nested datatypes in intensional type theory. *Science of Computer Programming*, 76(3):204–224, 2011. 15, 16

[23] Ralph Matthes and Tarmo Uustalu. Substitution in non-wellfounded syntax with variable binding. *Theor. Comput. Sci.*, 327(1-2):155–174, 2004. 2, 3, 4, 8, 9, 10, 11, 12, 14, 15, 16, 17, 19, 20, 22

[24] The Coq development team. *The Coq proof assistant reference manual*, 2015. Version 8.5beta2. 7

[25] Nax Paul Mendler. Inductive types and type constraints in the second-order lambda calculus. *Ann. Pure Appl. Logic*, 51(1-2):159–172, 1991. 9, 16

[26] Marino Miculan and Ivan Scagnetto. A framework for typed HOAS and semantics. In *PPDP*, pages 184–194. ACM, 2003. 11

[27] Erik Palmgren and Olov Wilander. Constructing categories and setoids of setoids in type theory. *Logical Methods in Computer Science*, 10(3):1 – 14, 2014. 6

[28] Matthieu Sozeau and Nicolas Tabareau. Universe polymorphism in Coq. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8558 of *Lecture Notes in Computer Science*, pages 499–514. Springer, 2014. 7

[29] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* `http://homotopytypetheory.org/book`, Institute for Advanced Study, 2013. 4, 5, 7

[30] Vladimir Voevodsky. C-system of a module over a monad on sets. 2014. `http://arxiv.org/abs/1407.3394`. 4

[31] Vladimir Voevodsky. An experimental library of formalized mathematics based on the univalent foundations. *Mathematical Structures in Computer Science*, 25:1278–1294, 6 2015. `http://arxiv.org/abs/1401.0053`. 7