Heterogeneous Substitution Systems Revisited

Benedikt Ahrens, Ralph Matthes

Heterogeneous Substitution Systems Revisited*

Benedikt Ahrens[1] and Ralph Matthes[2]

1 **Inria Rennes – Bretagne Atlantique,**
4 rue Alfred Kastler, 44307 Nantes Cedex 3, France
benedikt.ahrens@inria.fr

2 **Institut de Recherche en Informatique de Toulouse (IRIT),**
CNRS & Université Toulouse 3 Paul Sabatier,
118 route de Narbonne, 31062 Toulouse Cedex 9, France
matthes@irit.fr

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.

1998 ACM Subject Classification F.3.2 Logics and Meanings of Programs: Semantics of Programming Languages

Keywords and phrases formalization of category theory, nested datatypes, Mendler-style recursion schemes, representation of substitution in languages with variable binding

Digital Object Identifier 10.4230/LIPIcs.TYPES.2015.2

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

* The work of Benedikt Ahrens was partially supported by the CIMI (Centre International de Mathématiques et d'Informatique) Excellence program ANR-11-LABX-0040-CIMI within the program ANR-11-IDEX-0002-02 during a postdoctoral fellowship. This material is based upon work supported by the National Science Foundation under agreement Nos. 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.

(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 [9]. However, the monad laws by themselves do not express the (nearly) "homomorphic nature" of substitution.

In previous work, Matthes and Uustalu [24] 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. The authors then prove that any heterogeneous substitution system gives rise to a monad; multiplication of the monad is derived from the "substitution" operation.

Furthermore, it is shown there that, under some assumptions on the underlying category, "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. 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]) 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 [24]; 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 [24] extends functorially to morphisms. Moreover, we prove that the substitution system obtained in [24] 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 [12]. 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 type theory, more specifically, building upon the `UniMath` library [32]. This basis of our formalization is suitable in that it provides extensionality 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 [4] in `UniMath`.

1.1 Related work

Related work is extensively discussed in Matthes and Uustalu's article [24].

In the meantime, monads and modules over monads, have been used by Hirschowitz and Maggesi [18, 19] to define models of syntax, and to give a categorical characterization thereof.

The notion of signature introduced in [24] and formalized in the present work is similar to that employed in Hirschowitz and Maggesi's most recent work [17]. 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 [6, 5] on the construction of initial algebras in sets, however, this condition is of the essence.

Voevodsky [31] constructs a C-system from a module over a relative monad on sets, which in turn can be obtained from a monad on Set^2 and a choice of a set. Of particular interest as input to this construction are "term monads" generated by 2-sorted binding signatures. The present work does not directly allow for the construction of such monads. The follow-up work [5] describes a variant (alluded to in Remark 9) of the necessary results formalized in the current work that can be used for the construction of such monads.

1.2 Synopsis

In Section 2 we first give an overview of the system we work in: `UniMath`. Afterwards, we review the definition of categories in `UniMath`, 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 [24], 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 [24]: 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 [24] and our new results, have been formalized, based on the `UniMath` library [32]. More precisely, all results except for Theorem 22 and Lemmas 25 and 21 are proved in our formalization; Section 9 provides some technical details about our library.

2 Univalent Mathematics

The original article [24] 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 augmented by Voevodsky's *Univalence Axiom*. Specifically, we are working in the `UniMath` language and library, based on Voevodsky's *Foundations* [30].

2.1 About `UniMath`

`UniMath` is based on an intensional type theory augmented by Voevodsky's univalence axiom. In the following, we give a brief overview of the type constructors available in `UniMath`:

For a dependent type B over A there is the dependent pair type $\sum_{x:A} B(x)$, 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 $a = b$.

The Univalence Axiom identifies identities between types with equivalences between types, see [29, Axiom 2.10.3]. In this work, we do not use the full strength of the Univalence Axiom, but only function extensionality, a consequence of the Univalence Axiom.

In `UniMath`, there is an 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**.

Technically, the `UniMath` language is a subset of the language of the Coq proof assistant [13]: In order to simulate working in the theory described above, we do not use the full language Coq provides, but restrict ourselves to the language constructors mentioned there. 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 new 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 [30, Section 4]. To implement this resizing rule in Coq, we disable its checking of universes via a flag `-type-in-type` passed to the program. We hence work in a formally inconsistent system, and we have to check manually that we do not actually exploit that inconsistency.

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).

"Higher Inductive Types" (HITs), described, e. g., in the HoTT book [29], are not part of the axiomatically given type constructors of `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.

2.2 Category Theory in Univalent Type Theory

Category theory in univalent type theory has been developed in [4]. A category $\mathcal{C}$ is given by

- a type $\mathcal{C}_0$ of objects;
- for any $a, b : \mathcal{C}_0$, a set $\mathcal{C}(a, b)$ 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 $h \circ (g \circ f) = (h \circ g) \circ f$.

Note that we ask the hom-types $\mathcal{C}(a, b)$ of a category to be **sets**. This requirement enforces that the categorical axioms—which talk about equality of arrows—form propositions.

▶ Nota bene. There is an important difference between categories as usually formalized in intensional type theory and categories as considered in [4]: 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., [21]). These categories are sometimes referred to as "E-categories" [27].

In [4], 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.

In [4], an additional property of categories is studied: for any category $\mathcal{C}$, define a family of maps

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

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.

An important remark about naming: in [4], the univalence condition above is part of the definition of a category—the term "precategory" is employed for categories that are not necessarily univalent. That is, the authors of [4] use the terms "precategory" and "category" for what we call "category" and "univalent category" in the present article, respectively.

For the purposes of the present article, the univalence condition on categories is not essential. Indeed, no other result depends on Theorem 22. We thus choose to de-emphasize the importance of the univalence condition for categories by deviating from the naming of [4], and instead to make it explicit when considering categories that satisfy univalence.

3 Preliminaries

Categories, functors and natural transformations are defined in [4]. 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 $\mathsf{Ptd}(\mathcal{C})$ 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 $\mathsf{Ptd}(\mathcal{C})$ 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 $\mathsf{Ptd}(\mathcal{C})$. 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.

Note that the associator and unitor 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.

▶ **Remark 3.** In [24], the authors implicitly assume the monoidal structures of composition on $[\mathcal{C}, \mathcal{C}]$ and $\mathsf{Ptd}(\mathcal{C})$ to be strict. In univalent type theory, we have, e. g., that $F \cdot \mathsf{Id}$ is not convertible to F as a functor, but the two functors are convertible pointwise on objects and morphisms. This in turn entails that for $\rho_F : \mathsf{Id} \cdot F \to F$, the type $\rho_F = 1_F$ is well-typed. Note, however, that for an abstract functor on endofunctors H, the type $H(\rho_F) = H(1_F)$ is not well-typed.

In our definition of signatures (Definition 12) the associators and unitors do occur "under a functor application", where we cannot pretend (or even state) that they are identity morphisms. For reasons of symmetry, we hence decide to consider the monoidal structure of composition as non-strict, inserting the associator and unitors also in cases where this would not be necessary. In particular, we explicitly insert them in the strength laws of Definition 12 on the right-most position on the right hand side, respectively.

▶ **Definition 4** (Algebras of a functor)**.** For an endofunctor $F : \mathcal{C} \to \mathcal{C}$, the category $\mathsf{Alg}(F)$ 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 $f : \mathsf{Alg}(F)\big((X, \alpha), (X', \alpha')\big)$ is given by a morphism $f : \mathcal{C}(X, X')$ such that $f \circ \alpha = \alpha' \circ Ff$.

▶ **Remark 5.** We are using the arrow symbol "$\to$" for three different things:

1. 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);
2. functors $F : \mathcal{C} \to \mathcal{D}$ between categories; and
3. 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 [24] in taking monad multiplication μ as third component of a monad and not the Kleisli extension operation that is more widespread in computer science literature.

▶ **Definition 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 $\mathcal{D}$.

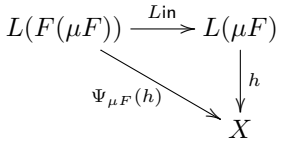
4 Generalized Iteration in Mendler-style and Fusion Law

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 [12] 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.

The link with the work by Mendler [25] is not made in the original proof [12, Thm. 2] of the lemma. The presentation in [12] is very much oriented towards functional programming. In their notation, the natural transformation Ψ would be typed as

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

▶ **Remark 9.** 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 ω-chains by F and L and the preservation of initial objects by L [12, Theorem 1]. We do not pursue that alternative in the present work.

In [24], 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 [24]). In fact, we do not need the fuller generality of generalized iteration in Mendler-style in Sections 7 and 8. However, the formulation of the fusion law to come next is more natural in the more general setting. No fusion law was needed in [24] 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 10** (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

$$\Phi_{\mu F}\big(\mathsf{It}^{L}_{F}(\,\Psi\,)\big) = \mathsf{It}^{L'}_{F}(\,\Psi'\,) \qquad \textit{if} \qquad \Phi_{F\mu F} \circ \Psi_{\mu F} = \Psi'_{\mu F} \circ \Phi_{\mu F}\ .$$

The name "fusion law" is wide-spread in functional programming for means to eliminate the creation of some extra datastructure. Here, the subsequent calculation of $\Phi_{\mu F}$ for the result $\mathsf{It}^{L}_{F}(\,\Psi\,)$ 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 [12] (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 freedom to choose 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—thus avoiding the need for 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. Instead, we 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 [24], 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).

▶ **Definition 11** (Relative strength). Let $(\mathcal{V}, \otimes, I)$ and $(\mathcal{W}, \bullet, E)$ be monoidal categories, and let

$$(U, \varphi, \varphi_0) : (\mathcal{W}, \bullet, E) \to (\mathcal{V}, \otimes, I)$$

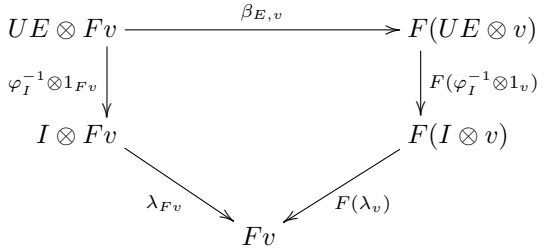
be a strong monoidal functor, that is, $\varphi_{w,w'} : Uw \otimes Uw' \cong U(w \bullet w')$ and $\varphi_I : I \cong UE$. Let $F : \mathcal{V} \to \mathcal{V}$ be a functor. A **tensorial strength for** F **relative to** (U, φ, φ_I) is a natural transformation

$$\beta_{w,v} : Uw \otimes Fv \to F(Uw \otimes v)$$

such that the following diagrams commute for any $w, w' : \mathcal{W}_0$ and $v : \mathcal{V}_0$:

$$\begin{array}{ccccc}
U(w \bullet w') \otimes Fv & \xrightarrow{\beta_{w \bullet w', v}} & F(U(w \bullet w') \otimes v) & \xrightarrow{F(\varphi^{-1}_{w,w'} \otimes 1_v)} & F((Uw \otimes Uw') \otimes v) \\
\downarrow{\scriptstyle \varphi^{-1}_{w,w'} \otimes 1_{Fv}} & & & & \downarrow{\scriptstyle F(\alpha_{Uw, Uw', v})} \\
(Uw \otimes Uw') \otimes Fv & & & & \\
\downarrow{\scriptstyle \alpha_{Uw, Uw', Fv}} & & & & \\
Uw \otimes (Uw' \otimes Fv) & \xrightarrow[1_{Uw} \otimes \beta_{w', v}]{} & Uw \otimes F(Uw' \otimes v) & \xrightarrow[\beta w, Uw' \otimes v]{} & F(Uw \otimes (Uw' \otimes v))
\end{array}$$

and



This definition is an instance of a broader definition of strength by Fiore [14, I.1.2]. Modulo order of arguments, our relative strength is a $\mathcal{W}$-strength of type $(\mathcal{V}, \odot) \to (\mathcal{V}, \odot)$, with the action $\odot$ induced by U, a construction that is also described by Fiore in the cited section.

▶ Nota bene. We find it important to mention two possible sources of confusion:

1. The notion of tensorial strength *relative to a strong monoidal functor* of Definition 11 is inspired by the notion of monad *relative to a functor* [7]. However, it is not the same as the concept of a *strength for a monad T relative to a functor $J : \mathcal{C} \to \mathcal{D}$*.
2. Note that the adjective "strong" is used in two different ways in the literature:
   - A *strong functor (or monad)* is a functor (or monad) equipped with a strength.
   - A *strong monoidal functor* is a monoidal functor for which the commutator morphisms $\varphi_{w,w'}$ and φ_I are isomorphisms, as recalled above.

We are interested in tensorial strengths for functors H relative to the forgetful functor $U : \mathsf{Ptd}(\mathcal{C}) \to [\mathcal{C}, \mathcal{C}]$ of Definition 1 that "forgets" the points of pointed functors. That particular functor is strict in the sense that φ and φ_I are identities. We hence set $(Z, e) \bullet (Z', e') := (Z' \cdot Z, e' \cdot e)$ and $X \otimes X' := X' \cdot X$ for the purpose of the following definition. Unfortunately, there is a mismatch between the order of the arguments of β in Definition 11 on the one hand—which is the order naturally arising when generalizing the traditional definition of strength—and the order in which Matthes and Uustalu [24] give the arguments to their instance of such a relative tensorial strength—called θ—in the following definition. We choose to retain compatibility with [24]:

▶ **Definition 12** (Signature)**.** Given a category $\mathcal{C}$, a **signature with strength** is a pair (H, θ) of an endofunctor H on $[\mathcal{C}, \mathcal{C}]$ and a tensorial strength for H relative to $U : \mathsf{Ptd}(\mathcal{C}) \to [\mathcal{C}, \mathcal{C}]$, that is, a natural transformation $\theta : (H-) \cdot U\sim \to H(- \cdot U\sim)$ between functors $[\mathcal{C}, \mathcal{C}] \times \mathsf{Ptd}(\mathcal{C}) \to [\mathcal{C}, \mathcal{C}]$ such that

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

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

We loosely refer to θ as the **strength of the signature** (H, θ).

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 with strength does not require the base category $\mathcal{C}$ to have coproducts, this is a requirement for most signatures with strength 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 signatures with strength, see Definition 15.

▶ **Convention 13.** 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 14.** The notion of signature with strength introduced in Definition 12 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 35 we discuss in detail in Section 8—that is not captured by the other works mentioned above.

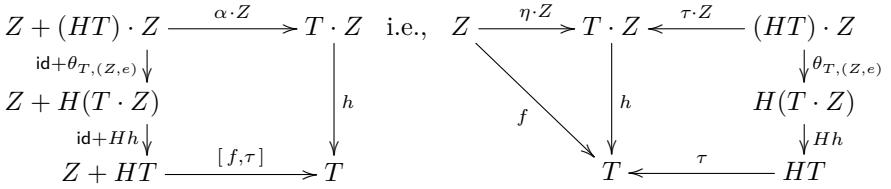
For a given signature (H, θ), we are interested in $(\underline{\mathsf{Id}} + H)$-algebras (T, α). For such an algebra, the natural transformation $\alpha : \mathsf{Id} + HT \to T$ decomposes into two $[\mathcal{C}, \mathcal{C}]$-morphisms $\eta : \mathsf{Id} \to T$, $\tau : HT \to T$ defined by

$$\eta = \alpha \circ \mathsf{inl}_{\mathsf{Id},HT} \qquad \text{and} \qquad \tau = \alpha \circ \mathsf{inr}_{\mathsf{Id},HT} \ . \tag{1}$$

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

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

▶ **Definition 15** (Def. 5 of [24], Heterogeneous substitution system of a signature)**.** We call (T, α) a *heterogeneous substitution system* for (H, θ), if, for every $\mathsf{Ptd}(\mathcal{C})$-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



For a heterogeneous 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 $\mathcal{C}$.

▶ Nota bene. Having freedom in the choice of parameter f (and its domain) is particularly important for Theorem 26, see Section 6. In its proof (not shown in this paper), 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.

In the following, we sometimes omit the word "heterogeneous" when talking about heterogeneous substitution systems. We refer to the operation $\{-\}$ by "substitution".

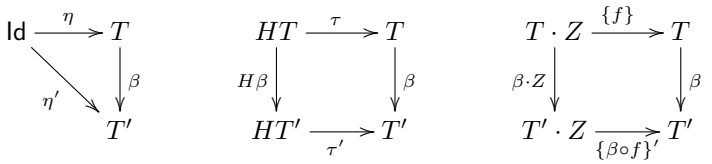
▶ **Remark 16.** Being equipped with a substitution operation $\{-\}$ is a proposition on $(\underline{\mathsf{Id}} + H)$-algebras.

The statement of the following lemma is mentioned, but not proven in [24]:

▶ **Lemma 17.** *The operation $\{-\}$ is a natural transformation*

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

▶ **Definition 18** (Category of substitution systems)**.** Given (H, θ) as before, the category $\mathsf{hss}(H, \theta)$ has, as objects, heterogeneous substitution systems as in Definition 15. A morphism of substitution systems is an algebra morphism that is compatible with the substitution $\{-\}$ on either side. In terms of η and τ as defined in Equation (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:



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 19.** Similarly to Remark 16, being compatible with the substitution 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 with strength 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 22: the category $\mathsf{hss}(H, \theta)$ is univalent if the base category $\mathcal{C}$ is.

Remarks 16 and 19 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 20.** 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, we 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 $g : \mathcal{C}(b, c)$, we 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 16 and 19. The resulting category is clearly isomorphic to $\mathsf{hss}(H, \theta)$ in the sense of [4, Definition 6.9].

Note that isomorphic categories are propositionally equal [4, 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 21.** *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 substitution $\{-\}'$ 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 15,

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

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

$$\{\beta^{-1} \circ f\} = \beta^{-1} \circ h \circ \beta \cdot Z \ \ , \tag{2}$$

which follows from the uniqueness of $\{-\}$: it suffices to show that the right-hand side of (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. ◀

▶ **Theorem 22.** *The category* $\mathsf{hss}(H, \theta)$ *is univalent if* $\mathcal{C}$ *is.*

Proof. Combine Lemmas 23, 25, 24 below and Lemma 21 above. 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 the category of $(\underline{\mathsf{Id}} + H)$-algebras. ◀

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

▶ **Lemma 23.** *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. ◀

The next lemma is originally due to Hofmann and Streicher [20]; and is also proved in Thm. 4.5 of [4]:

▶ **Lemma 24.** *The category of functors* $[\mathcal{C}, \mathcal{D}]$ *is univalent if the target category* $\mathcal{D}$ *is.*

The category of substitution systems 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 25.** *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 $\mathsf{idtoiso}$. ◀

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

6 From Substitution Systems to Monads

One of the most important results of Matthes and Uustalu's work [24] is the construction of a monad from any substitution system:

▶ **Theorem 26** ([24], 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 27.** *The map from heterogeneous substitution systems to monads defined in [24, 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 substitution systems, 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 [24, Thm. 10]. This is an easy consequence of the compatibility of β with $\{\}$ and $\{\}'$. ◀

▶ Nota bene. Hirschowitz and Maggesi [17] observe that any signature with strength (H, θ) yields a module transformer: given a module (M, ρ) over a monad R, then HM is again a module over R. The module multiplication of HM is defined as $H\rho \circ \theta_{M,R}$.

Given a heterogeneous substitution system $(T, [\eta, \tau], \{-\})$, the monad $(T, \eta, \{\mathsf{id}_{(T,\eta)}\})$ (from now on denoted just T) constructed in Theorem 26 can be viewed as a module over itself. Applying the aforementioned module transformer yields a module (with underlying functor) HT over T.

Further along those lines, we note that $\tau : HT \to T$ is a module morphism in the sense of [19] between the modules over T thus defined. The equation establishing this is an instance of the family of equations on heterogeneous substitution systems ruling the case of constructors—that is, of the family of squares concerning the morphism τ in Definition 15: the instance where $(Z, e) := (T, \eta)$ and $f := \mathsf{id}_{(T,\eta)}$.

One might thus say that being a substitution system includes by definition that τ is a module morphism. However, this only concerns the equation to be fulfilled. It does not suggest a modification of the notion of substitution systems: when defining substitution systems, the modules for which τ is a morphism are not available, and thus, it could not even be stated in the definition of substitution systems that τ ought to be a morphism between these modules.

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 28.** *The functor of Theorem 27 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. ◀

▶ **Remark 29.** The functor of Theorem 27 is not full. For instance, choose $\mathcal{C} = \mathsf{Set}$, and take a signature with two copies app and app' (of the same arity) of an "application" constructor, see Definition 33 in Section 8. Take the initial substitution system associated to that signature with strength (as constructed via Theorems 30 and 31 in Section 7), and define

an endomorphism on it that maps app to 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. 18 does not commute—any endomorphism on that substitution system must be the identity morphism.

▶ Nota bene. The question may arise if we could modify our results on the construction of monads to obtain relative monads [7]. However, non-relativized monads are the more general outcome, since, by composing the constituents of a monad (in the presentation with Kleisli extension instead of a monad multiplication) with a given functor J, we would obtain monads relative to J (called "restriction" [7, Proposition 2.3(1)]).

7 Lifting Initiality Through a Fusion Law

The starting point of this section is a result from [24], 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 [12], and (the carriers of) the corresponding initial F-algebras are called "nested datatypes" [10]. 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 can be defined syntactically: the syntactic right Kan extension of type transformer G along type transformer Z is defined as $\lambda A \forall B.\,(A \to ZB) \to GB$, which is monotone in A for syntactic reasons regardless of G and Z (A occurs non-strictly positively in the body of the abstraction). This construction is essential for relating different formulations of iteration over nested datatypes [3]. However, the full categorical properties of Kan extensions are not ensured by the computation rules of the polymorphic language. Still, they are satisfied in parametric models of higher-order polymorphism [16, Thm. 6.10 (i)]. 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 [23] (using Coq very differently from the `UniMath` approach).

▶ **Theorem 30** ([24], 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 $\mathsf{Ptd}(\mathcal{C})$-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

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

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, using the construction that extends solutions for parameter X naturally to solutions for parameter FX, the lemma even yields a (unique) solution for the least fixed-point of F as parameter. The continuity properties behind this method were deeply explored by Abel [2] for (co-)inductive types and extended to nested datatypes later [1].

This is the essence of schemes in Mendler's style [25]: being able to advance 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 [24].[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 strength θ of the signature and the H-algebra τ that is generically derived from α (see before Definition 15).

▶ Nota bene. We remark that all of this is not optimal from a progammer's point of view, where the question is not only of soundness but of efficiency of the traversals through the data structures. There is the more refined notion of "generalized Mendler iteration" [3] (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 [3].

Also for generalized Mendler iteration, there is a formulation in more conventional terms of algebras, called "generalized refined conventional iteration" [3], which captures in particular the efficient folds of Martin, Gibbons and Bayley [22]. 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 [23].

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

[1] The instantiation with $- \cdot Z$ for L can also be formulated in a less homogeneous setting where not only endofunctor categories intervene [24, Section 2.3].

▶ **Theorem 31.** *The substitution system* $(T, \alpha, \{\})$ *constructed in Lemma 30 is initial in* $\mathsf{hss}(H, \theta)$.

In order to prove Theorem 31, 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 30) and $\{\}'$. That is, we need to show that, for any $f : (Z, e) \to (T, \eta)$,

$$! \circ \{f\} = \{! \circ f\}' \circ (! \cdot Z) \ . \tag{3}$$

Using the fusion law (Lemma 10), we show that both sides of (3) 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 premisses of the fusion law established, we can show equality with the right-hand side of (3) 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 [24] but originates from [8, 11].

We can "glue" signatures with strength together to obtain a new signature with strength:

▶ **Lemma 32** (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 [24] on λ-calculus without and with a form of explicit substitution—"explicit flattening". In order to do so, we first present the signatures (H, θ) corresponding to application, abstraction, and explicit flattening, respectively:

▶ **Definition 33** (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 strength 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 34** (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 35** (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., [3]). 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 32, to obtain the signatures we are mainly interested in:

▶ **Definition 36** (Signature of λ-calculus)**.** The signature Λ is obtained as the sum of the signatures of Defs. 33 and 34.

▶ **Definition 37** (Signature of λ-calculus with explicit flattening)**.** The signature Λ^{μ} is obtained as the sum of the signatures of Defs. 36 and 35.

For the purpose of this example, we assume the signatures Λ and Λ^{μ} to have initial substitution systems. By Lemma 30 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 equation 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 integrated into the given term. In other words, μ_B removes the "cross section" between the trunk of the term and the term-like variable leaves. Invoking Theorem 26 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 TB (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. Hence, the constructor is of type $\mathsf{Lam}^\mu \cdot \mathsf{Lam}^\mu \to \mathsf{Lam}^\mu$, which is the same type as monad multiplication. 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 therefore called an explicit flattening operation. (Cf. explicit substitution; in fact, explicit flattening is a variant of explicit substitution.) Already in [24], 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 substitution 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 15.

Once this is done, we obtain, by initiality, a morphism of substitution systems from the initial substitution system 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 38** (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 39** (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 substitution $\{\}^{\mathsf{Flatten}}$ *on this algebra by setting, for* (Z, e) *and* $f : (Z, e) \to (\mathsf{Lam}, \eta)$*,*

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

This assignment defines substitution 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 substitution, see Definition 15. 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 39, with underlying carrier Lam. By initiality, we obtain a unique morphism of substitution systems in this category.

▶ **Definition 40.** 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 substitution systems gives rise, via functoriality of the monad construction (Theorem 27), to a monad morphism; it is this morphism that is studied in Example 16 of [24]. 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 [32]. More precisely, all results except for Theorem 22 and Lemmas 25 and 21 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 32;
`LamSignature.v` corresponds to Definitions 33, 34, 35;
`Lam.v` corresponds to the rest of Section 8.

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

To account for the ongoing work on the `UniMath` library, we provide an "interface" file `UniMath/SubstitutionSystems/SubstitutionSystems_Summary.v` containing pointers to the most important formalized theorems. So, while this paper is best studied with an eye on the commit mentioned above, the interface file allows to locate all the notions, constructions and results in their respective current state of evolution.

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

[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.

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.

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 make opaque all the terms that we could afford making opaque, 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 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 multiple stacked constructions in category theory: 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 [6]. 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.

Added in print: the use of primitive projections in Coq, via the option `Set Primitive Projections`, reduced the compilation time for our project dramatically (much better than the 56% drop in time that were observed for the whole `UniMath` library on average). This was adopted for the Σ-types used in `UniMath` in `commit 6b044cc`.

10 Conclusions

We presented, in univalent type theory, some new results about the heterogeneous substitution systems introduced by Matthes and Uustalu [24], 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, in the present work, the construction of initial algebras in univalent type theory; this is the subject of joint work with Mörtberg [6, 5].

Acknowledgements 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 A. Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. Doktorarbeit (PhD thesis), LMU München, 2006.

2 Andreas Abel. Termination checking with types. *ITA*, 38(4):277–319, 2004. doi:10.1051/ita:2004015.

3 Andreas Abel, Ralph Matthes, and Tarmo Uustalu. Iteration and coiteration schemes for higher-order and nested datatypes. *Theor. Comput. Sci.*, 333(1-2):3–66, 2005. doi:10.1016/j.tcs.2004.10.017.

4 B. Ahrens, K. Kapulkin, and M. Shulman. Univalent categories and the Rezk completion. *Math. Struct. Comput. Sci.*, 25(5):1010–1039, 2015. doi:10.1017/s0960129514000486.

5 B. Ahrens, R. Matthes, and A. Mörtberg. From signatures to monads in UniMath, 2016. arXiv preprint 1612.00693. URL: https://arxiv.org/abs/1612.00693.

6 Benedikt Ahrens and Anders Mörtberg. Some wellfounded trees in UniMath - extended abstract. In Gert-Martin Greuel, Thorsten Koch, Peter Paule, and Andrew J. Sommese, editors, *Mathematical Software - ICMS 2016 - 5th International Conference, Berlin, Germany, July 11-14, 2016, Proceedings*, volume 9725 of *Lecture Notes in Computer Science*, pages 9–17. Springer, 2016. doi:10.1007/978-3-319-42432-3_2.

7 T. Altenkirch, J. Chapman, and T. Uustalu. Monads need not be endofunctors. *Log. Methods Comput. Sci.*, 11(1):article 3, 2015. doi:10.2168/lmcs-11(1:3)2015.

8 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, 8th Annual Conference of the EACSL, Madrid, Spain, September 20-25, 1999, Proceedings*, volume 1683 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 1999. doi:10.1007/3-540-48168-0_32.

9 Françoise Bellegarde and James Hook. Subsitution: A formal methods case study using monads and transformations. *Sci. Comput. Program.*, 23(2-3):287–311, 1994. doi:10.1016/0167-6423(94)00022-0.

10 R. Bird and L. Meertens. Nested datatypes. In J. Jeuring, editor, *Proc. of 4th Int. Conf. on Mathematics of Program Construction, MPC '98*, volume 1422 of *Lect. Notes in Comput. Sci.*, pages 52–67. Springer, 1998. doi:10.1007/bfb0054285.

11 R. S. Bird and R. Paterson. De Bruijn notation as a nested datatype. *J. Funct. Program.*, 9(1):77–91, 1999. doi:10.1017/s0956796899003366.

12 Richard S. Bird and Ross Paterson. Generalised folds for nested datatypes. *Formal Asp. Comput.*, 11(2):200–222, 1999. doi:10.1007/s001650050047.

13 The Coq Development Team. The Coq proof assistant reference manual, version 8.6, 2016. URL: https://coq.inria.fr/distrib/current/refman/.

14 M. Fiore. Second-order and dependently-sorted abstract syntax. In *Proc. of 23rd Ann. IEEE Symp. on Logic in Computer Science, LICS 2008*. IEEE, 2008. doi:10.1109/lics.2008.38.

15 M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In *Proc. of 14th Ann. IEEE Symp. on Logic in Computer Science, LICS '99*, pages 193–202. IEEE, 1999. doi:10.1109/lics.1999.782615.

16 Ryu Hasegawa. Parametricity of extensionally collapsed term models of polymorphism and their categorical properties. In Takayasu Ito and Albert R. Meyer, editors, *Theoretical Aspects of Computer Software, International Conference TACS '91, Sendai, Japan, September*

24-27, 1991, Proceedings, volume 526 of *Lecture Notes in Computer Science*, pages 495–512. Springer, 1991. doi:10.1007/3-540-54415-1_61.

17 A. Hirschowitz and M. Maggesi. Initial semantics for strengthened signatures. In D. Miller and Z. Ésik, editors, *Proc. of 8th Wksh. on Fixed Points in Computer Science, FICS 2012*, volume 77 of *Electron. Proc. in Theor. Comput. Sci.*, pages 31–38. Open Publishing Assoc., 2012. doi:10.4204/eptcs.77.5.

18 André Hirschowitz and Marco Maggesi. Modules over monads and linearity. In Daniel Leivant and Ruy J. G. B. de Queiroz, editors, *Logic, Language, Information and Computation, 14th International Workshop, WoLLIC 2007, Rio de Janeiro, Brazil, July 2-5, 2007, Proceedings*, volume 4576 of *Lecture Notes in Computer Science*, pages 218–237. Springer, 2007. doi:10.1007/978-3-540-73445-1_16.

19 André Hirschowitz and Marco Maggesi. Modules over monads and initial semantics. *Inf. Comput.*, 208(5):545–564, 2010. doi:10.1016/j.ic.2009.07.003.

20 M. Hofmann and T. Streicher. The groupoid interpretation of type theory. In G. Sambin and J. M. Smith, editors, *Twenty-Five Years of Constructive Type Theory*, volume 36 of *Oxford Logic Guides*, pages 127–172. Clarendon Press, 1998.

21 G. Huet and A. Saïbi. Constructive category theory: Essays in honour of Robin Milner. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction*, Foundations of Computing Series, pages 239–275. MIT Press, 2000.

22 Clare E. Martin, Jeremy Gibbons, and Ian Bayley. Disciplined, efficient, generalised folds for nested datatypes. *Formal Asp. Comput.*, 16(1):19–35, 2004. doi:10.1007/s00165-003-0013-6.

23 Ralph Matthes. Map fusion for nested datatypes in intensional type theory. *Sci. Comput. Program.*, 76(3):204–224, 2011. doi:10.1016/j.scico.2010.05.008.

24 Ralph Matthes and Tarmo Uustalu. Substitution in non-wellfounded syntax with variable binding. *Theor. Comput. Sci.*, 327(1-2):155–174, 2004. doi:10.1016/j.tcs.2004.07.025.

25 N. P. Mendler. Inductive types and type constraints in the second-order lambda calculus. *Ann. Pure Appl. Log.*, 51(1–2):159–172, 1991. doi:10.1016/0168-0072(91)90069-x.

26 Marino Miculan and Ivan Scagnetto. A framework for typed HOAS and semantics. In *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden*, pages 184–194. ACM, 2003. doi:10.1145/888251.888269.

27 E. Palmgren and O. Wilander. Constructing categories and setoids of setoids in type theory. *Log. Methods Comput. Sci.*, 10(3):article 25, 2014. doi:10.2168/lmcs-10(3:25)2014.

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. doi:10.1007/978-3-319-08970-6_32.

29 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013. URL: http://homotopytypetheory.org/book.

30 V. Voevodsky. An experimental library of formalized mathematics based on the univalent foundations. *Math. Struct. Comput. Sci.*, 25:1278–1294, 2015. doi:10.1017/s0960129514000577.

31 V. Voevodsky. C-system of a module over a Jf-relative monad, 2016. arXiv preprint 1602.00352. URL: https://arxiv.org/abs/1602.00352.

32 V. Voevodsky, B. Ahrens, D. Grayson, and Others. UniMath: Univalent mathematics. URL: https://github.com/UniMath.