Explicit Substitutions and Higher-Order Syntax

Neil Ghani[1] and Tarmo Uustalu[2]

[1] Dept. of Math. and Comp. Sci., University of Leicester
University Road, Leicester LE1 7RH, UK, ng13@mcs.le.ac.uk
[2] Institute of Cybernetics, Tallinn Technical University
Akadeemia tee 21, EE-12618 Tallinn, Estonia, tarmo@cs.ioc.ee

Abstract. Recently there has been a great deal of interest in higher-order syntax which seeks to extend standard initial algebra semantics to cover languages with variable binding by using functor categories. The canonical example studied in the literature is that of the untyped λ-calculus which is handled as an instance of the general theory of binding algebras, cf. Fiore, Plotkin, Turi [8]. Another important syntactic construction is that of explicit substitutions. The syntax of a language with explicit substitutions does not form a binding algebra as an explicit substitution may bind an arbitrary number of variables. Nevertheless we show that the language given by a standard signature Σ and explicit substitutions is naturally modelled as the initial algebra of the endofunctor $\mathsf{Id} + F_\Sigma \circ \_ + \_ \circ \_$ on a functor category. We also comment on the apparent lack of modularity in syntax with variable binding as compared to first-order theories.

1 Introduction

Initial algebra semantics has long been regarded as one of the cornerstones of the semantics of programming languages. Within this paradigm, the syntax of a language is modelled as an initial algebra, consisting of the terms of the language while the semantics of the language is typically given as the unique homomorphism from the initial algebra into some other algebra. A similar situation arises in the theory of datatypes, where the initial algebra consists of the terms built from the constructors of the datatype while initiality allows functions to be defined over the datatype via structural recursion. The success of this program has also motivated the recent development of final coalgebra semantics for non-wellfounded syntax.

Categorically, one regards such an initial algebra as the initial algebra of an endofunctor F specifying the language or datatype. In order to incorporate *variables*, *substitution*, one considers not a single initial algebra, but rather, for each object X, the initial $(X + F\_)$-algebra, which is the free F-algebra over X. The mapping sending an object X, thought of as an object of variables, to (the carrier of) the initial $(X+F\_)$-algebra is the underlying functor of the free *monad* over F. The multiplication of the monad provides an abstract representation of substitution, the unit models the variables, while the freeness of the monad models the inductive nature of the initial algebra. Applications to base categories other than **Set** have proved fruitful in many situations, e.g. the study of S-sorted algebraic theories as monads over $\mathbf{Set}^S$, the study of categories with structure using monads over **Graph** or **Cat** [6], the study of rewriting using monads over **Pre** or **Cat** [14].

It has long been a goal of theoreticians and language designers to incorporate datatypes involving variable binding into this framework. The classic example here is that of the untyped λ-calculus which plays a foundational role in computer science, e.g., as a theory of computability or as a paradigmatic functional programming language. Intuitively, the problem is to define an inductive family of types rather than just a family of individually inductive types. By observing that a family of types is just an object of a functor category, Fiore, Plotkin and Turi [8] showed that *binding algebras* could indeed be defined by deploying initial algebra semantics in functor categories.

From a computational point of view, reduction in the λ-calculus is usually modelled by β-reduction. However this has several defects from an implementational point of view. Most strikingly, β-reduction may duplicate redexes. There have been several attempts to overcome this problem, e.g., research on optimal reduction and term graph rewriting. Another option is to introduce *explicit substitutions* into the syntax of the language—computation is then split into many finer, local reductions which gives the implementer more flexibility as to when and where redexes are actually contracted.

This paper asks the question: Can we understand explicit substitutions via initial algebra semantics? A positive answer to this question is important, if we are to prove correct the implementation of languages based upon explicit substitutions. While [8] provides the right setting and ideas in suggesting the use of functor categories, the actual functor we are interested in is not a binding algebra in that the explicit substitution operator can bind any number of free variables. This paper solves the problem as follows.

- We argue that the syntax of a language given by a standard signature and explicit substitutions should be the initial algebra of a certain very simple endofunctor on a functor category. This gives associated soundness and completeness results for the inference rules for explicit substitutions.
- We show that this endofunctor does indeed have an initial algebra and, using [18], show that the explicit substitutions functor is a monad as described above. We also show how the canonical evaluation map arises naturally.
- We suggest that, unlike first-order theories, higher-order syntax is not modular in the sense that one cannot decompose the monad capturing the syntax of a large language as the coproduct of the monads corresponding to the components of the language.

In Section 2, we review the necessary preliminaries, which are lfp categories and Kan extensions. In Section 3, we review the categorical approach to the initial algebra semantics of first-order syntax. In Section 4, we look at the untyped λ-calculus as the paradigmatic example of a language with variable binding and switch to initial algebras in functor categories. The solution for explicit substitutions is presented in Section 5 which constitutes the main section of the paper. The problem with modularity in higher-order syntax is discussed in Section 6. In Section 7, we list some conclusions and ideas regarding future work.

2 Preliminaries

We assume the reader is familiar with standard category theory as can be found in [17]. In order to ensure that our results are combinator based, and hence easier to implement, we abstract away from the category **Set** to *locally finitely presentable categories* and make passing use of *coends* and *Kan extensions*. We give a short summary of the important results concerning these concepts while referring the reader to the literature for more details [17, 4]. Alternatively, we provide significant intuition for all our constructions to allow the reader to follow the thrust of our argument for the category **Set**.

Locally Presentable Categories One important feature of explicit substitutions and other similar syntactic constructs is the role of finite sets, e.g., contexts consist of finite sets of variables and terms contain finite numbers of variables. In fact every set is the directed join of its finite subsets. To abstract these properties, we will work in locally finitely presentable categories.

A diagram $\mathcal{D}$ is *filtered* iff every finite subcategory of $\mathcal{D}$ has a cocone in $\mathcal{D}$. A functor is *finitary* iff it preserves filtered colimits. An object X of a category $\mathcal{A}$ is said to be *finitely presentable* iff the hom-functor $\mathcal{A}(X, \_)$ preserves filtered colimits. A category is *locally finitely presentable* (abbreviated as lfp) iff it is cocomplete and has a set N of finitely presentable objects such that every object is a filtered colimit of objects from N. The full subcategory of finitely presentable objects is denoted $\mathcal{A}_f$. The inclusion functor is denoted $I_f : \mathcal{A}_f \to \mathcal{A}$ and the category of finitary functors from $\mathcal{A}$ to $\mathcal{B}$ is denoted $[\mathcal{A}, \mathcal{B}]_f$.

If $\mathcal{A} = \mathbf{Set}$, the finitely presentable sets are precisely finite sets, N can be taken to be the finite cardinals, which we denote $\mathbb{F}$, and every set is the filtered colimit of the diagram of all its finite subsets (each of which is isomorphic to a finite cardinal) ordered by inclusion.

Kan Extensions Given a functor $I : \mathcal{A} \to \mathcal{B}$ and a category $\mathcal{C}$, precomposition with I defines a functor $\_ \circ I : [\mathcal{B}, \mathcal{C}] \to [\mathcal{A}, \mathcal{C}]$. The problem of left and right Kan extensions is the problem of finding left and right adjoints to $\_ \circ I$. More concretely, given a functor $F : \mathcal{A} \to \mathcal{C}$, the left and right Kan extensions of F along I are characterized by the natural isomorphisms

$$[\mathcal{B}, \mathcal{C}](\mathsf{Lan}_I F, H) \cong [\mathcal{A}, \mathcal{C}](F, H \circ I) \quad [\mathcal{B}, \mathcal{C}](H, \mathsf{Ran}_I F) \cong [\mathcal{A}, \mathcal{C}](H \circ I, F).$$

Thus, one can view the left and right Kan extension of a functor $F : \mathcal{A} \to \mathcal{C}$ along $I : \mathcal{A} \to \mathcal{B}$ as the canonical extensions of the domain of F to $\mathcal{B}$.

Kan extensions can be given pointwise using colimits and limits, or more elegantly using ends and coends. We will use the classic coend formula for left Kan extensions (see [17] for details)

$$\mathsf{Lan}_I F(X) = \int^{a:\mathcal{A}} \mathcal{B}(Ia, X) \otimes Fa$$

where $\otimes$ is the tensor (copower) of $\mathcal{C}$. In this paper, coends reduce to certain colimits, which again reduce to quotients. As we mentioned earlier, we will give intuitions behind the use of such formulae so readers without knowledge in this area can follow the argument.

Importantly, if $\mathcal{A}$ is lfp, then a functor $F : \mathcal{A} \to \mathcal{B}$ is finitary iff it is isomorphic to $\mathsf{Lan}_{I_f}(F \circ I_f)$.

3 Free Algebras and Free Monads

The general program we wish to follow is that of specifying or declaring syntax by means of a functor with the actual terms of the associated language arising via an initial algebra construction. Furthermore, these terms will be parameterised by variables they are built over, which should imbue the terms with a natural notion of substitution axiomatised by the concept of a monad. Since the notions of algebra and monad are therefore central to our program, we begin with their definition.

Algebras of an Endofunctor Let $F : \mathcal{C} \to \mathcal{C}$ be a functor. An *F-algebra* is a pair (X, h) where X is in $\mathcal{C}$ and $h : FX \to X$. The category of F−**alg** has as objects F-algebras and a map from the F-algebra (X, h) to the F-algebra (X', h') is a map $f : X \to X'$ such that $f \cdot h = h' \cdot Ff$.

The simplest cases of practical interest are those polynomial endofunctors arising over a signature. A *signature* is a functor $\Sigma : |\mathbb{F}| \to \mathbf{Set}$ and the object of *n-ary operations* of Σ is $\Sigma_n = \Sigma n$. For such a signature, one defines the polynomial endofunctor either concretely by $F_\Sigma X = \coprod_{n \in |\mathbb{F}|} X^{Jn} \times \Sigma_n$ or more elegantly by $F = \mathsf{Lan}_J \Sigma$ where $J : |\mathbb{F}| \to \mathbf{Set}$ is the inclusion. Either way, F_Σ computes the terms of depth 1 built from Σ over a set of variables. An F_Σ-algebra is then the standard notion of a model for Σ, i.e., a set X and an interpretation for each constructor in the signature as an operator over X.

Monads A *monad* $T = (T, \eta, \mu)$ on a category $\mathcal{C}$ is an endofunctor $T : \mathcal{C} \to \mathcal{C}$ together with two natural transformations, $\eta : \mathsf{Id}_\mathcal{C} \to T$, called the *unit*, and $\mu : T \circ T \to T$, called the *multiplication* of the monad, satisfying the monad laws $\mu{\cdot}T\eta = \mathsf{Id}_\mathcal{C} = \mu{\cdot}\eta_T$, and $\mu{\cdot}T\mu = \mu{\cdot}\mu_T$.

The category $\mathbf{Mon}(\mathcal{C})$ has as objects monads on $\mathcal{C}$ and a map from a monad $T = (T, \eta, \mu)$ to $T' = (T', \eta', \mu')$ is a natural transformation $f : T \to T'$ satisfying $f \cdot \eta = \eta'$ and $f \cdot \mu = \mu' \cdot (f \circ f)$.

A monad is said to be *finitary* iff its underlying endofunctor is finitary. $\mathbf{Mon}_\mathrm{f}(\mathcal{C})$ is the full subcategory of finitary monads.

The Term Algebra of a Signature The Σ-term algebra (= the Σ-syntax) over a set of variables X is generated by the rules

$$\frac{x \in X}{x \in T_\Sigma X} \qquad \frac{f \in \Sigma_n \quad t_1, \dots t_n \in T_\Sigma X}{f(t_1, \dots, t_n) \in T_\Sigma X}$$

Two observations are immediate. First, the set $T_\Sigma X$ is, for any set X, the carrier an $\mathsf{Id} + F_\Sigma$-algebra. Second, the set endofunctor T_Σ is also the underlying functor of a monad. The unit maps a variable to the associated term, while the multiplication describes the process of substitution. The monad laws ensure that substitution behaves correctly, i.e., substitution is associative and the variables are left and right units. This demonstrates that monads form an abstract calculus for equational reasoning about *variables, substitution* and *term algebra* (represented by the unit, multiplication and underlying functor of the monad).

Both observations, however, are weak: the fail to capture the inductive nature of T_Σ, which is crucial in understanding its behavior. Resolving this problem is crucial to extending to our understanding to syntax with variable binding. To this end, we reformulate the set-theoretic definition.

We note that every term is either a variable or a term constructor applied to a variable, that is, $T_\Sigma X$ solves the equation $Z \cong X + F_\Sigma Z$. Moreover, it is actually the least solution. Hence, $T_\Sigma X$ should be the initial $(X + F_\Sigma \_)$-algebra.

Lemma 1. *Let $\mathcal{C}$ be an ω-cocomplete category $\mathcal{C}$ and F an ω-cocontinuos functor. Then there is an initial F-algebra and its carrier is computable as the colimit of the ω-chain J where $J_0 = 0$, $J_{n+1} = F J_n$.*

Lemma 2. *Let $\mathcal{C}$ be a category with finite coproducts and F an endofunctor on $\mathcal{C}$. Then the initial $(X + F\_)$-algebras (TX, α_X), if they exist, have the following properties:*

1. *(TX, α_X) is the free F-algebra over X, i.e., the functor $X \mapsto (TX, \alpha_X)$: $\mathcal{C} \to F{-}\mathbf{alg}$ is left adjoint to the forgetful functor $U : F{-}\mathbf{alg} \to \mathcal{C}$,*
2. *$X \mapsto TX$ is the underlying functor of the free monad over F, i.e., T carries a monad structure and there is natural transformation $h : F \to T$ such that for any monad S and natural transformation $h' : F \to S$, there is a unique monad map $f : T \to S$ satisfying $f \cdot h = h'$.*

Applying Lemma 1 to functors F_Σ on **Set** arising from a signature, we need to establish the following to guarantee the existence the initial $(X + F_\Sigma \_)$-algebras.

- **Set** has the required colimits. Actually, **Set** is cocomplete.
- F_Σ is ω-cocontinuous. F_Σ is a polynomial endofunctor, i.e., built from $+$ and $\times$. In any CCC such as **Set**, $+$ and $\times$ are left adjoint and hence preserve all colimits.

Applying Lemma 2, we see that $T_\Sigma X$ is the carrier of the free F_Σ-algebra over X and T_Σ is the underlying functor of the free monad over F_Σ.

In practice, we tend to require that $\mathcal{C}$ is lfp and F finitary. Clearly every monad has an underlying functor while we have just shown that, for a finitary functor on an lfp category, we can construct a free monad over it. These constructions are adjoint.

Lemma 3. *The forgetful functor $U : \mathbf{Mon}_{\mathrm{f}}(\mathcal{C}) \to [\mathcal{C},\mathcal{C}]_f$ has a left adjoint sending a functor F to the free monad on it.*

Since left adjoints preserve colimits, $T_{\Sigma+\Sigma'} \cong T_\Sigma + T_{\Sigma'}$ (as this is in $\mathbf{Mon}_{\mathrm{f}}(\mathcal{C})$, the $+$ sign stands for coproduct of monads, not of functors), which means that we can decompose large syntactic structures into their components. As we comment later, it is intriguing that the same does hold for higher-order syntax.

There are various generalisations of this work which are worth pointing out. Operations with larger arity can be handled by moving to locally κ-presentable categories [11]. Non-wellfounded syntax can be treated by replacing various initial algebra constructions by final coalgebra constructions [21, 1, 11]. Other forms of syntax, e.g., Σ-term graphs and Σ-rational terms can also be generalised to universal constructions involving F_Σ-algebras on lfp categories [2, 20, 3, 10].

4 Untyped λ-calculus

Can we extend the above analysis to languages with variable binding? Many proposals have been made, but one of the best is that of [8, 9] (cf. also [12, 5]) whose elegance relies in the fact that one uses exactly the same general program as for first-order languages but instantiated in functor categories.

For example, if LX is the set of untyped λ-terms up to α-equivalence with free variables from a (typically finite) set X, then it ought to be that

$$LX \cong X + LX \times LX + L(1 + X)$$

In the right-hand side, the first summand represents variables, the second summand represents applications while the third represents λ-abstractions. Note how the 1+ part of the expression means there is an extra variable—this is exactly the bound variable. Furthermore, the sets LX should be the least solution of the above equation so as to permit definition of functions by structural recursion. This should be obtained by using an initial algebra construction.

However, there is a problem. With first-order theories, we code, for each X, the right-hand side of the equation up as a functor and construct the term algebra over X by proving the existence of an initial algebra for this functor. However, λ-abstraction introduces an extra variable and so we cannot construct each set LX separately. Instead we must construct all sets LX simultaneously and this is achieved by moving to the functor category $[\mathbf{Set},\mathbf{Set}]_f$. More abstractly, we can use $[\mathcal{C},\mathcal{C}]_f$ where $\mathcal{C}$ is lfp. We want to construct a finitary endofunctor L on $\mathcal{C}$ such that

$$LX \cong \mathsf{Id}X + (\Delta L)X + (\delta L)X$$

where $\Delta, \delta : [\mathcal{C},\mathcal{C}]_f \to [\mathcal{C},\mathcal{C}]_f$ (with $\mathcal{C}$ possibly $\mathbf{Set}$) are defined by $(\Delta F)X = FX \times FX$, $(\delta F)X = 1 + X$. Since colimits in functor categories are computed pointwise, this is equivalent to

$$LX \cong (\mathsf{Id} + \Delta L + \delta L)X \quad \text{i.e.,} \quad L \cong \mathsf{Id} + \Delta L + \delta L$$

Now we can use the techniques of the previous section. Define a functor $\Lambda : [\mathcal{C}, \mathcal{C}]_f \to [\mathcal{C}, \mathcal{C}]_f$ by

$$\Lambda X = \mathsf{Id} + \Delta X + \delta X$$

and L will arise as the carrier of the initial Λ-algebra. In order to ensure the existence of the initial Λ-algebra, by Lemma 1, it suffices to check the following:

- $[\mathcal{C}, \mathcal{C}]_f$ has finite coproducts and ω-colimits. This holds, because they are inherited from $\mathcal{C}$, which we assume to be lfp.
- The functor Λ preserves ω-colimits. This reduces to showing that Δ and δ preserve ω-colimits, which is a routine verification.

Recall that apart from expecting the syntax of a language to be an initial algebra construction, we also want it to be a monad to ensure that it comes with a well-behaved operation of substitution and to be able to formulate the requirement that semantic functions respect substitution. We have constructed L as the carrier of the initial algebra of the functor $\Lambda : [\mathcal{C}, \mathcal{C}]_f \to [\mathcal{C}, \mathcal{C}]_f$. We will show that it is a monad at the same time as proving that the syntax of a first-order language with explicit substitutions forms a monad in the next section. We note however that L is not a free monad over an endofunctor.

To conclude, we can use initial algebra semantics in a functor category to define inductive families of types such as the untyped λ-terms. The fact that no new mathematics is required to make the theory work is a really compelling argument in favour of the (categorical formulation of) initial algebra semantics. Handling the syntax of simply typed λ-calculus is possible as well [5, 7, 19]

5 Explicit Substitutions

The core of this paper asks whether the above treatment extends to explicit substitutions. This is certainly a reasonable idea since explicit substitutions involve variable binding and we have seen how elegantly variable binding can be modelled in functor categories.

Note however there is a potential pitfall in that unlike λ-calculi, the explicit substitution operation binds an unknown number of variables. Thus a priori it is not clear how to write for explicit substitutions an equation like the one for λ-terms in the previous section, where we knew that λ-abstraction binds exactly one variable. The explicit substitutions operator has a flexible binding power. We present an elegant way of tackling this problem which thereby allows us to bring into play the standard initial algebra techniques described above.

Let us assume we are interested in adding explicit substitutions to a first-order theory—our argument clearly scales up adding explicit substitutions to λ-calculus. Thus we start with a signature and generate terms according to the rules

$$\frac{x \in X}{x \in E_\Sigma X} \qquad \frac{f \in \Sigma_n \quad t_1, \ldots t_n \in E_\Sigma X}{f(t_1, \ldots, t_n) \in E_\Sigma X} \qquad \frac{\theta : I_f n \to E_\Sigma X \quad t \in E_\Sigma(I_f n)}{sub(\theta, t) \in E_\Sigma X}$$

where I_f is the inclusion $\mathbb{F} \to \mathbf{Set}$. So, $E_\Sigma X$ is generated exactly as the first-order terms $T_\Sigma X$, but there is an additional inference rule which says that if we have a term built over n variables, and a mapping of each of these variables into terms (possibly containing explicit substitutions) over a given set X, then we form an explicit substitution by binding the variables in n to their associated terms. Thus the set $E_\Sigma X$ depends upon $E_\Sigma(I_f n)$ and we have the same problem as we faced with the untyped calculus: we are dealing not with a family of inductive types, but with an inductive family of types. Intuitively, we may try writing something like

$$E_\Sigma X \cong X + F_\Sigma(E_\Sigma X) + \coprod_{n \in |\mathbb{F}|} (E_\Sigma X)^{I_f n} \times E_\Sigma(I_f n)$$

However, this does not account for α-equivalence. We have to ask ourselves when should an explicit substitution given by $\theta : I_f n \to E_\Sigma X, t \in E_\Sigma(I_f n)$ be equal to an explicit substitution given by $\theta' : I_f n' \to E_\Sigma X, t' \in E_\Sigma(I_f n')$. The approach taken by this paper is to stipulate the following rule:

$$\frac{f : n \to n' \quad \theta' : I_f n' \to E_\Sigma X \quad t \in E_\Sigma(I_f n)}{sub(\theta' \cdot I_f f, t) = sub(\theta', (E_\Sigma f)t)}$$

Expressing this quotient of $E_\Sigma X$ by the above equivalence relation is very elegantly done using a coend. Let us abstract away from signatures to functors; let $\mathcal{C}$ be lfp and F a finitary endofunctor on $\mathcal{C}$. Then we want that E_F is a finitary endofunctor on $\mathcal{C}$ and

$$E_F X \cong X + F(E_F X) + \int^{Y:\mathcal{C}_f} \mathcal{C}(I_f Y, E_F X) \otimes E_F(I_f Y)$$

where I_f is now the inclusion $\mathcal{C}_f \to \mathcal{C}$. When $\mathcal{C} = \mathbf{Set}$, the tensor is just the product as expected. Also notice that we have taken the coend over the full subcategory of finitely presentable objects, which corresponds to our observation that we are dealing with finite contexts and hence the explicit substitutions operator binds a finite number of bound variables.

One could now try to construct E_F by defining a higher-order functor to capture the right-hand side and proving that this it preserves ω-colimits. While possible, the presence of a coend makes this a delicate task. A simpler idea is to use some abstract category theory to rearrange the above formula into a simpler form. Note first that the coend above is actually computing the left Kan extension $\mathsf{Lan}_{I_f}(E_F \circ I_f)(E_F X)$. In addition, we certainly have that E_F will be finitary and so $\mathsf{Lan}_{I_f}(E_F \circ I_f) \cong E_F$.[3] Thus we may instead proceed from

$$E_F X \cong X + F(E_F X) + E_F(E_F X)$$

[3] Note that from here it follows that although we employed a coend over $\mathcal{C}_f$, we could have just as well used a coend over the whole of $\mathcal{C}$, viz. $\int^{Y:\mathcal{C}} \mathcal{C}(Y, E_F X) \otimes E_F Y$, since that would have have computed $\mathsf{Lan}_{\mathsf{Id}_\mathcal{C}} E_F$, which of course is just E_F as well. But the smaller coend is the intuitively right thing a priori; the adequacy of the bigger coend is an a posteriori fact.

Switching to the functor category, we obtain

$$E_F \cong \mathsf{Id} + F \circ E_F + E_F \circ E_F$$

Hence, we should look for the initial algebra of $\Psi_F : [\mathcal{C}, \mathcal{C}]_f \to [\mathcal{C}, \mathcal{C}]_f$ given by

$$\Psi_F X = \mathsf{Id} + F \circ X + X \circ X$$

Does the initial Ψ_F-algebra exist? We know that $[\mathcal{C}, \mathcal{C}]_f$ has finite coproducts and ω-colimits inherited from $\mathcal{C}$ so we must ask whether Ψ_F is ω-cocontinuous. This boils down to asking whether the functor $\_ \circ \_ : [\mathcal{C}, \mathcal{C}]_f \to [\mathcal{C}, \mathcal{C}]_f$ is ω-cocontinuous, which follows from direct calculation. Clearly, the same argument works, if we want to add explicit substitutions to the untyped λ-calculus rather than just a first-order theory.

So we know that we can use functor categories to get an initial algebra semantics for languages with explicit substitutions. We finally turn to the question of whether we always get a monad. For this, we use the following theorem from [18]. [The paper cited actually gives a considerably more general theorem, which, also applies to, e.g., non-wellfounded syntax, but we need only a special case.]

Theorem 1. *Let $\mathcal{C}$ be a category with finite coproducts such that $\mathsf{Ran}_Z X$ exists for all $X, Z : \mathcal{C} \to \mathcal{C}$, Z pointed. Given a functor $\Phi : [\mathcal{C}, \mathcal{C}] \to [\mathcal{C}, \mathcal{C}]$ and a natural transformation $\theta : (\Phi \__1) \circ \__2 \to \Phi(\__1 \circ \__2)$ between functors $[\mathcal{C}, \mathcal{C}] \times \mathbf{Ptd}(\mathcal{C}) \to [\mathcal{C}, \mathcal{C}]$ satisfying*

$$\begin{aligned} \theta_{X,\mathsf{Id}} &= \mathsf{id}_{\Phi X} \\ \theta_{X,Z' \circ Z} &= \theta_{X \circ Z', Z} \cdot (\theta_{X,Z'})_Z \end{aligned}$$

the carrier of the initial $(\mathsf{Id} + \Phi \_)$-algebra, if this exists, is a monad.

Now to see that L is a monad, we set $\Phi X = X \times X + X \circ (1+)$ for which we can use

$$\begin{aligned} \theta_{X,Z} &= \mathsf{id}_{X \circ Z \times X \circ Z} + X \circ [\eta^Z_{1+} \cdot \mathsf{inl}_{1,\mathsf{Id}}, Z \circ \mathsf{inr}_{1,\mathsf{Id}}] \\ &: \; X \circ Z \times X \circ Z + X \circ (1+) \circ Z \to X \circ Z \times X \circ Z + X \circ Z \circ (1+) \end{aligned}$$

For E_F, we set $\Phi X = F \circ X + X \circ X$ and may rely on

$$\begin{aligned} \theta_{X,Z} &= \mathsf{id}_{F \circ X \circ Z} + X \circ \eta^Z_{X \circ Z} \\ &: \; F \circ X \circ Z + X \circ X \circ Z \to F \circ X \circ Z + X \circ Z \circ X \circ Z \end{aligned}$$

Remarkably, the monad structure on L resp. E_F explicated by this theorem is the intended one in the sense that the substitution operation we get from the theorem coincides with the normal capture-avoiding substitution of λ-terms resp. F-terms with explicit substitutions.

Evaluation of explicit substitutions One important feature of explicit substitutions is that any term with explicit substitutions can be normalised or reduced to one without explicit substitutions, and this respects variables and substitution. In our framework this corresponds to defining a natural transformation $\mathsf{eval} : E_F \to T_F$ and verifying that it is a monad map.

This is very simply described in our initial algebra setting. Firstly note that since E_F is the carrier of the initial Ψ_F-algebra, the evaluation map can be given by structural recursion which takes imbuing T_F with a Ψ_F-algebra structure. This means we need a natural transformation $h : (\mathsf{Id} + F \circ T_F) + T_F \circ T_F \to T_F$. This is $h = [\alpha, \mu]$ where α is the initial algebra structure on T_F whereas μ is the multiplication on T_F.

Checking that eval is a monad map is easy. Hence pure F-terms qualify as a model for F-terms with explicit substitutions as a syntax, and evaluation qualifies as a semantic function.

6 Non-Modularity of Higher-Order Syntax

We mentioned in Section 3 that if F and G were finitary endofunctors on an lfp category, with T_F and T_G the associated free monads, then $T_{F+G} \cong T_F + T_G$. This result means the representing monad of a composite first-order signature can be decomposed into the representing monads of the component signatures. More concretely, a semantic function (a map respecting variables and substitution) from a large syntax T_{F+G} to a model (a set carrying variables and substitution) is completely determined by its restrictions to two smaller syntaxes T_F and T_G. Intuitively, the reason is that any $(F+G)$-term over X is constructible in a wellfounded fashion from F-terms and G-terms over X and substitution (in fact, one only needs variables, depth-1 F-terms and depth-1 G-terms). This and other applications of coproducts of monads as a modularity tool have been discussed in [16, 15].

Similar decomposition theorems for higher-order syntax fail, i.e., coproducts of monads do not make higher-order syntax modular, not alone at least.

Specifically, one might maybe expect that $E_F \cong T_F + E$ where $E = \mu(\mathsf{Id} + \_ \circ \_)$, which would say that extending first-order syntax with explicit substitutions amounts to taking its coproduct with pure explicit substitution terms. This would be a nice result in that one could regard explicit substitutions as a kind of computational monad which one could combine with other forms of syntax by taking coproducts. But this is false in general! Also untrue is $L \cong L_{\mathrm{app}} + L_{\mathrm{abs}}$ where $L_{\mathrm{app}} = \mu(\mathsf{Id} + \_ \times \_)$ and $L_{\mathrm{abs}} = \mu(\mathsf{Id} + \_ \circ (1+))$ are the languages of pure application terms and pure abstraction terms.

To see this, we invoke the following theorem [13].

Theorem 2. *Given an endofunctor F and a monad S on $\mathcal{C}$, if the free monads over F and $F \circ S$ exist, then the coproduct of S and T_F exists and*

$$T_F + S \cong S \circ T_{F \circ S} \cong \mu(S \circ (\mathsf{Id} + F \circ \_))$$

From this theorem, we see that

$$T_F + E \cong \mu(E \circ (\mathsf{Id} + F \circ \_))$$

Unwinding, we get

$$T_F + E \cong E \circ (\mathsf{Id} + F \circ (E \circ (\mathsf{Id} + F \circ (E \circ ...))))$$

where

$$E \cong \mathsf{Id} + (\mathsf{Id} + (\ldots)^2)^2$$

Hence, the elements of $T_F + E$ are like the usual F-terms except being padded with layers of explicit substitutions, but these may only have pure explicit substitution terms as stem terms. It is not possible to explicitly substitute into a term involving F-operations, i.e., a subterm of the form $sub(\theta, t)$ is illegal, if t involves F-operations. In contrast, E_F allows arbitrary mixes of explicit substitutions and F-operations in its elements. We see that $T_F + E$ does not contain enough terms and hence is not as interesting as E_F which is clearly the real object of study.

Similarly, we obtain that

$$L_{\mathrm{app}} + L_{\mathrm{abs}} \cong \mu(L_{\mathrm{abs}} \circ (\mathsf{Id} + \_ \times \_))$$

Unwinding and distribution give

$$L_{\mathrm{app}} + L_{\mathrm{abs}} \cong L_{\mathrm{abs}} \circ (\mathsf{Id} + (L_{\mathrm{abs}} \circ ...) \times (L_{\mathrm{abs}} \circ ...))$$

where

$$\begin{aligned} L_{\mathrm{abs}} &\cong \mathsf{Id} + (\mathsf{Id} + (\mathsf{Id} + \ldots) \circ (1+)) \circ (1+)) \\ &\cong \mathsf{Id} + \mathsf{Id} \circ (1+) + \mathsf{Id} \circ (2+) + \ldots \end{aligned}$$

The elements of $L_{\mathrm{app}} + L_{\mathrm{abs}}$ are therefore like pure applicative terms (terms built of variables and application only) padded with layers of abstractions, but: the choice between the new and the old variables is made in a wrong place. In the body of an abstraction which is not another abstraction, it is first decided where the new variable or the old ones will be used. As a result, the body of an abstraction can be an application if the application only uses the old variables: subterms $\lambda y.st$ where y occurs free in s or t are forbidden.

Can we interpret this? Yes: For E_F to be the coproduct of T_F and E, every monad map $f : E_F \to S$ has to be uniquely specified by its restrictions to T_F and E. But there are elements in E_F which cannot be constructed from elements in T_F and E only by substitution (the proper substitution of E_F, which never substitutes into the stem of an explicit substitution, since all its free variables are bound by the explicit substitution), e.g., terms of the form $sub(\theta, t)$ where t involves F-operations. Knowing only the restrictions of f to T_F and E and that the fact that f respects substitution, it is thus impossible to decide the value of f on such terms.

Similarly, L is not the coproduct of L_{app} and L_{abs} because not every λ-term can be constructed from pure application and pure abstraction terms using the proper, capture-avoiding substitution of L: no terms of the form $\lambda y.st$ where y actually occurs free in s or t are constructible in this way. (The only possibility would be $\lambda y.st = (\lambda y.x)[x \mapsto st]$, but this violates the freeness for substitution proviso.) If only the restrictions of a monad map $f : L \to S$ to L_{app} and L_{abs} are known, then absolutely nothing can be inferred about the value of f on $(\lambda y.xy)$!

In conclusion: In the coproduct combination of the syntaxes of two languages, the terms of the combined syntax are constructed from terms of each given syntax using the proper substitution of the resulting syntax which obeys the substitution laws. In combinations like the λ-calculus syntax or a first-order syntax plus explicit substitutions, the combined syntax is generated from the given ones by means of some naive pseudo-substitution.

7 Conclusions and Future Work

Research on higher-order syntax has the potential to produce really exciting developments in language design. However there are a number of key problems which need to be solved before the potential benefits can be realised. Of those that interest us, there are two that we are currently working on.

- Modularity in syntax is important. We do not want to have to redevelop the metatheory of a language every time we change a component. As we discussed in Sec. 5, the simple approach to modularity based upon taking coproducts of monads fails in the higher-order setting. Yet clearly there is modularity here and we would like to obtain a mathematical analysis of this.
- We also want to understand the model theory of higher-order datatypes. In the first-order setting, we know that there is a bijection between monad maps $T_F \to S$ and natural transformations $F \to S$. If we generate representing monads for higher-order functors using Theorem 1, then can we classify the monad maps from the representing monad to another monad? Similar, intimately related questions arise for the algebras of the functor and representing monad.

Acknowledgements N. Ghani's research was partially supported by EPSRC under grant No. GR/M96230/01. His visit to IoC in Feb. 2003 was financed by the Estonian Ministry of Education and Research through the Center for Dependable Computing. T. Uustalu received support from the Estonian Science Foundation under grant No. 5567. His visit to Leicester in March 2003 was paid by the Estonian IT Foundation.

References

1. P. Aczel, J. Adámek, S. Milius, and J. Velebil. Infinite trees and completely iterative theories: A coalgebraic view. *Theor. Comp. Sci.*, 300(1–3):1–45, 2003.
2. J. Adámek, S. Milius, and J. Velebil. Free iterative theories: A coalgebraic view. *Math. Struct. in Comp. Sci.*, 13(2):259–320, 2003.

3. J. Adámek, S. Milius, and J. Velebil. On rational monads and free iterative theories. In R. Blute and P. Selinger, eds., *Proc. of 9th Conf. on Category Theory and Comp. Sci., CTCS'02*, v. 69 of *Electr. Notes in Theor. Comp. Sci.* Elsevier, 2003.
4. J. Adámek and J. Rosický. *Locally Presentable and Accessible Categories*, v. 189 of *London Math. Soc. Lecture Note Series.* Cambridge Univ. Press, 1994.
5. T. Altenkirch and B. Reus. Monadic presentations of lambda terms using generalized inductive types. In J. Flum and M. Rodríguez-Artalejo, eds., *Proc. of 13th Int. Wksh. on Comp. Sci. Logic, CSL'99*, v. 1683 of *Lect. Notes in Comp. Sci.*, pp. 453–468. Springer-Verlag, 1999.
6. E. J. Dubuc and G. M. Kelly. A presentation of topoi as algebraic relative to categories or graphs. *J. of Algebra*, 83:420–433, 1983.
7. M. Fiore. Semantic analysis of normalisation by evaluation for typed lambda calculus. In *Proc. of 4th Int. ACM SIGPLAN Conf. on Principles and Practice of Declarative Programming, PPDP'02*, pp. 26–37. ACM Press, 2002.
8. M. Fiore, G. D. Plotkin, and D. Turi. Abstract syntax and variable binding (extended abstract). In *Proc. of 14th Ann. IEEE Symp. on Logic in Comp. Sci., LICS'99*, pp. 193–202. IEEE CS Press, 1999.
9. M. Fiore and D. Turi. Semantics of name and value passing. In *Proc. of 16th Ann. IEEE Symp. on Logic in Comp. Sci., LICS'01*, pp. 93–104. IEEE CS Press, 2001.
10. N. Ghani, C. Lüth, and F. de Marchi. Coalgebraic monads. In L. S. Moss, ed., *Proc. of 5th Wksh. on Coalgebraic Methods in Comp. Sci., CMCS'02*, v. 65(1) of *Electr. Notes in Theor. Comp. Sci.* Elsevier, 2002.
11. N. Ghani, C. Lüth, F. de Marchi, and J. Power. Dualising initial algebras. *Math. Struct. in Comp. Sci.*, 13(2):349–370, 2003.
12. M. Hofmann. Semantical analysis of higher-order abstract syntax. In *Proc. of 14th Ann. IEEE Symp. on Logic in Comp. Sci., LICS'99*, pp. 204–213. IEEE CS Press, 1999.
13. M. Hyland, G. Plotkin, and J. Power. Combining computational effects: Commutativity and sum. In A. Baeza-Yates, U. Montanari, and N. Santoro, eds., *Proc. of IFIP 17th World Computer Congress, TC1 Stream / 2nd IFIP Int. Conf. on Theor. Comp. Sci., TCS 2002*, v. 223 of *IFIP Conf. Proc.*, pp. 474–484. Kluwer Acad. Publishers, 2002.
14. C. Lüth and N. Ghani. Monads and modular term rewriting. In E. Moggi and G. Rosolini, eds., *Proc. of 7th Int. Conf. on Category Theory and Comp. Sci., CTCS'97*, v. 1290 of *Lect. Notes in Comp. Sci.*, pp. 69–86. Springer-Verlag, 1997.
15. C. Lüth and N. Ghani. Composing monads using coproducts. In *Proc. of 7th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'02*, v. 37(9) of *SIGPLAN Notices*, pp. 133–144. ACM Press, New York, 2002.
16. C. Lüth and N. Ghani. Monads and modularity. In A. Armando, ed., *Proc. of 4th Int. Wksh. on Frontiers of Combining Systems, FroCoS 2002*, v. 2309 of *Lect. Notes in Artif. Intell.*, pp. 18–32. Springer-Verlag, 2002.
17. S. Mac Lane. *Categories for the Working Mathematician*, v. 5 of *Graduate Texts in Math.* Springer-Verlag, 2nd ed., 1997. (1st ed., 1971).
18. R. Matthes and T. Uustalu. Substitution in non-wellfounded syntax with variable binding. In H. P. Gumm, ed., *Proc. of 6th Int. Wksh. on Coalg. Methods in Comp. Sci., CMCS'03*, v. 82(4) of *Electr. Notes in Theor. Comp. Sci.* Elsevier, 2003.
19. M. Miculan and I. Scagnetto. A framework for typed HOAS and semantics. To appear in Proc. of PPDP'03, ACM Press, 2003.
20. S. Milius. On iteratable endofunctors. In R. Blute and P. Selinger, eds., *Proc. of 9th Conf. on Category Theory and Comp. Sci., CTCS'02*, v. 69 of *Electr. Notes in Theor. Comp. Sci.* Elsevier, 2003.
21. L. S. Moss. Parametric corecursion. *Theor. Comp. Sci.*, 260(1–2):139–163, 2001.