Explicit Substitutions and Higher-Order Syntax

Neil Ghani
School of Computer Science and IT, University of Nottingham,
Jubilee Campus, Wollaton Rd, Nottingham NG8 1BB, UK, nxg@cs.nott.ac.uk

Tarmo Uustalu
Institute of Cybernetics, Tallinn University of Technology,
Akadeemia tee 21, EE-12618 Tallinn, Estonia, tarmo@cs.ioc.ee

Makoto Hamana
Department of Computer Science, Gunma University,
1-5-1 Tenjin-cho, Kiryu, Gunma 376-8515, Japan, hamana@cs.gunma-u.ac.jp

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. 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 [13].

Another important syntactic construction is that of explicit substitutions which are used to model local definitions and to implement reduction in the λ-calculus. 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. Thus explicit substitutions are a natural test case for the further development of the theory and applications of syntax with variable binding.

This paper shows that a language containing explicit substitutions and a first-order signature Σ is naturally modelled as the initial algebra of the $\mathsf{Id} + F_\Sigma \circ \_ + \_ \circ \_$ endofunctor. We derive a similar formula for adding explicit substitutions to the untyped λ-calculus and then show these initial algebras provide useful datatypes for manipulating abstract syntax by implementing two reduction machines. We also comment on the apparent lack of modularity in syntax with variable binding as compared to first-order languages.

Keywords: abstract syntax, variable binding, explicit substitutions, algebras, monads

1. Introduction

Abstract Syntax

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-well-founded 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* and *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** [11], the study of rewriting using monads over **Pre** or **Cat** [19].

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 [13] showed that *binding algebras* could indeed be defined by deploying initial algebra semantics in functor categories, i.e., initial algebras of *higher-order* functors.

Explicit Substitutions

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 can be seen in both the $\lambda\sigma$-calculus [1] and in reduction machines such as Krivine's machine [10].

This paper asks the question: Can we understand explicit substitutions via initial algebra semantics? And can we use this theoretical understanding to provide datatypes to support high-level implementations of algorithms involving explicit substitutions? While [13] 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 first-order 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.
- Using [23], we show that the explicit substitutions functor is a monad as described above. We also show how, as with the $\lambda\sigma$-calculus, explicit substitutions can be evaluated away. The usual presentation of explicit substitutions as a quotient datatype means evaluation is complicated by the usual problems with α-equivalence etc. However, the presentation via a higher-order initial algebra significantly simplifies this.
- We also consider reduction machines as they appear in the literature and show how our higher-order approach to explicit substitutions allows a very straightforward implementation without the usual problems concerning bound variables etc. Again, it is the presentation of explicit substitutions as a free datatype rather than a quotient datatype, and the support in Haskell for higher-order type constructors and rank-2 type signatures, which permits this.
- We finish by suggesting that, unlike first-order languages, 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.

This paper

This paper is an extended version of [17] and builds upon [23], which contained a key theorem about when the initial algebra of an endofunctor on a functor category forms a substitution system and thus a monad, and mentioned, as an example, two versions of the explicit

substitution operator. This paper discusses the example in detail showing that the two versions are equivalent and do really capture explicit substitutions as we know them from the literature. In addition, it uses the coend-based presentation of explicit substitutions to implement reduction machines and thereby demonstrates the practical usefulness of our results. Consequently, this paper will appeal to both theoreticians who are interested in the further development of higher-order abstract syntax and to more applied readers who can see how abstract syntax can be used to program with. To further ensure the breadth of this paper, we have also included a summary of initial algebra semantics and of the Altenkirch-Reus and Fiore-Plotkin-Turi approach to variable binding [7, 13].

A terminological remark: Throughout this paper, speaking of higher-order syntax, we refer to accounts of syntax in terms of initial algebras of higher-order functors. It is important to realize that this is different from higher-order abstract syntax (HOAS) in the sense of describing syntax with variable binding via a logical framework.

Organization
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 central section of the paper. In Section 6, we give Haskell implementations of the untyped λ-calculus syntax, of explicit substitutions, and of two reduction machines. The problem with modularity in higher-order syntax is discussed in Section 7. In Section 8, 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 [22]. 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 [22, 6]. Alternatively, we provide sig-

nificant 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 category $\mathcal{D}$ 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 some finitely presentable objects such that every object is a filtered colimit of objects from N. The full subcategory of all finitely presentable objects is denoted $\mathcal{A}_{\mathrm{f}}$. The inclusion functor is denoted $I_{\mathrm{f}} : \mathcal{A}_{\mathrm{f}} \to \mathcal{A}$ and the category of finitary functors from $\mathcal{A}$ to $\mathcal{B}$ is denoted $[\mathcal{A}, \mathcal{B}]_{\mathrm{f}}$.

If $\mathcal{A} = \mathbf{Set}$, the finitely presentable sets are precisely finite sets, N can be taken to be $|\mathbb{F}|$, where $\mathbb{F}$ is the category of finite cardinals, 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

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

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 [22] for details)

$$(\mathsf{Lan}_I F)B = \int^{A:\mathcal{A}} \mathcal{B}(I\,A, B) \otimes F\,A$$

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 $\mathsf{Lan}_{I_\mathrm{f}}(F \circ I_\mathrm{f})$.

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 the variables they are built over. This will 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 operators* 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 of finite cardinals (without functions between them) into sets. 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 operator in the signature as an operation 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 $\mu \cdot T\eta = \mathsf{id}_T = \mu \cdot \eta_T$, 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-language) over a set of variables X is generated by the rules

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

Two observations are immediate. First, the set $T_\Sigma X$ is, for any set X, the carrier of an $(X + F_\Sigma{}_-)$-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: they fail to capture the inductive nature of T_Σ, which is crucial in understanding its behaviour. 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 an operator applied to terms, that is, $T_\Sigma X$ solves the equation $Y \cong X + F_\Sigma Y$. Moreover, it is actually the least solution. Hence, $T_\Sigma X$ should be the initial $(X + F_\Sigma{}_-)$-algebra.

The following results are standard.

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

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 for all X, have the following properties:*

1. *(TX, α_X) is a free F-algebra over X, i.e., the functor $X \mapsto (TX, \alpha_X) : \mathcal{C} \to F$-**alg** is left adjoint to the forgetful functor $U : F$-**alg** $\to \mathcal{C}$.*

2. *T is the underlying functor of a 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 $f \cdot h = h'$.*

3. *(T, α) is an initial $(\mathsf{Id} + F \circ \_)$-algebra.*

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 a free F_Σ-algebra over X and T_Σ is the underlying functor of a 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. *Let $\mathcal{C}$ be an lfp category. Then the forgetful functor $U : \mathbf{Mon}_{\mathrm{f}}(\mathcal{C}) \to [\mathcal{C},\mathcal{C}]_{\mathrm{f}}$ 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 second $+$ 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 not 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 [16]. Non-wellfounded syntax can be treated by replacing various initial algebra constructions by final coalgebra constructions [27, 3, 16]. 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 [4, 26, 5, 15].

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 [7, 13, 14], 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 (de Bruijn terms) 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 for λ-abstractions means there is an extra variable—this is exactly the bound variable (as we work with de Bruijn terms, bound variables are nameless). 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 languages, 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 $[\mathbf{Set}, \mathbf{Set}]_{\mathrm{f}}$. More abstractly, we can use $[\mathcal{C}, \mathcal{C}]_{\mathrm{f}}$ where $\mathcal{C}$ is lfp. We want to construct a finitary endofunctor L on $\mathcal{C}$ such that

$$LX \cong X + (\Delta L)X + (\delta L)X$$

where $\Delta, \delta : [\mathcal{C}, \mathcal{C}]_{\mathrm{f}} \to [\mathcal{C}, \mathcal{C}]_{\mathrm{f}}$ (with $\mathcal{C}$ possibly $\mathbf{Set}$) are defined by $(\Delta F)X = FX \times FX$, $(\delta F)X = F(1 + X)$. 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 $\Lambda : [\mathcal{C}, \mathcal{C}]_{\mathrm{f}} \to [\mathcal{C}, \mathcal{C}]_{\mathrm{f}}$ by

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

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:

- $[\mathcal{C}, \mathcal{C}]_{\mathrm{f}}$ has 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 $\Lambda : [\mathcal{C}, \mathcal{C}]_{\mathrm{f}} \to [\mathcal{C}, \mathcal{C}]_{\mathrm{f}}$. We will show that it is a monad at the same time as proving that λ-terms with explicit substitutions also form 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 [7, 12, 25].

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 operator 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 substitution 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 language—we add explicit substitutions to λ-calculus later. Thus we start with a signature Σ and generate terms according to the rules

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

$$\frac{\theta : I\, n \to E_\Sigma X \quad t \in E_\Sigma (I\, n)}{sub(\theta, t) \in E_\Sigma X}$$

where I 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 (a body), and a mapping of each of these variables into terms over a given set X (a substitution rule), then we form an explicit substitution term by

binding the variables in n to their associated terms. Thus the set $E_\Sigma X$ depends upon $E_\Sigma(In)$ 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

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

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

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

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

$$E_F X \cong X + F(E_F X) + \int^{W:\mathcal{C}_\mathrm{f}} \mathcal{C}(I_\mathrm{f} W, E_F X) \otimes E_F(I_\mathrm{f} W)$$

where, recall, I_f is the inclusion $\mathcal{C}_\mathrm{f} \to \mathcal{C}$. 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 $\mathsf{Lan}_{I_\mathrm{f}}(E_F \circ I_\mathrm{f})(E_F X)$. In addition, we certainly have that E_F will be finitary and so $\mathsf{Lan}_{I_\mathrm{f}}(E_F \circ I_\mathrm{f}) \cong E_F$.[1] Thus we may instead proceed from

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

[1] Note that from here it follows that although we employed a coend over $\mathcal{C}_\mathrm{f}$, we could have just as well used a coend over the whole of $\mathcal{C}$, viz. $\int^{W:\mathcal{C}} \mathcal{C}(W, E_F X) \otimes E_F W$, since that would have have computed $\mathsf{Lan}_{\mathsf{Id}_\mathcal{C}} E_F(E_F X)$, where $\mathsf{Lan}_{\mathsf{Id}_\mathcal{C}} E_F$ 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 $\Psi_F : [\mathcal{C},\mathcal{C}]_{\mathrm{f}} \to [\mathcal{C},\mathcal{C}]_{\mathrm{f}}$ given by

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

Does the initial Ψ_F-algebra exist? We know that $[\mathcal{C},\mathcal{C}]_{\mathrm{f}}$ 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 $\_ \circ \_ : [\mathcal{C},\mathcal{C}]_{\mathrm{f}} \to [\mathcal{C},\mathcal{C}]_{\mathrm{f}}$ 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 language. That is, to consider a language consisting of the untyped λ-calculus with explicit substitutions, one would look for a functor E which is the least solution of

$$E \cong \mathsf{Id} + \delta E + \Delta E + E \circ E$$

where the summands correspond the variables, abstractions, applications and explicit substitutions.

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 [23]. [The paper cited gives a considerably more general theorem, which also applies, e.g., to non-wellfounded syntax, but we need only a special case.] Let $\mathbf{Ptd}(\mathcal{C})$ denote the category of pointed endofunctors over $\mathcal{C}$ (with the unit of a pointed functor Z denoted by η^Z).

THEOREM 1. *Let $\mathcal{C}$ be a category with finite coproducts such that $\mathsf{Ran}_Z Y$ exists for all $Y, Z : \mathcal{C} \to \mathcal{C}$, 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*

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

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

The definition of the monad structure can be found in [23]. The conditions on θ are exactly those of a strength of H (wrt. functorial composition), except that we only require $\theta_{Y,Z}$ to be defined if Z is pointed. Instead of existence of right Kan extensions, it is enough to require that the initial algebra is strong in a suitable sense.

Now to see that the functor for untyped λ-terms, denoted above by L, is a monad, we set $\Phi Y = Y \times Y + Y \circ (1+)$. Then

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

For explicit substitutions over a first-order signature F, we set $\Phi Y = F \circ Y + Y \circ Y$ and may rely on

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

As mentioned above, for the untyped λ-calculus with explicit substitutions, we set $\Phi Y = Y \times Y + Y \circ (1+) + Y \circ Y$ and combine the arguments for L and E_F.

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 evaluated to one without explicit substitutions, and this respects variables and substitution. In our framework this corresponds to defining a natural transformation $\mathsf{resolve} : E_F \to T_F$ 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 entails 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 $(\mathsf{Id}+F\circ\_)$-initial algebra structure on T_F whereas μ is the multiplication on T_F.

Checking that $\mathsf{resolve}$ 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. This definition of evaluation is considerably simpler than those appearing in the literature, e.g., on evaluation for the $\lambda\sigma$-calculus. This is because our mathematics has given us a higher-order fold combinator from the initial algebra to any other algebra which automatically handles the problems of α-equivalence.

6. Untyped λ-calculus and Explicit Substitutions in Haskell

We now proceed to showing that the modelling of the untyped lambda calculus syntax and explicit substitution operator presented in Sections 4 and 5 is not only elegant in its categorical version, but is also straightforwardly implementable in the functional language Haskell [28] using some of its elegant non-standard advanced features such as rank-2 type signatures [24]. We implement the syntax of untyped λ-calculus both without and with explicit substitutions and then, as example applications, implement two classic reduction machines taken from [10].

The Haskell code of this section is available from the web at `http://cs.ioc.ee/~tarmo/haskell/expsubst/`.

We start by recalling that monads (in the Kleisli format) are built in to Haskell on the level of Prelude definitions as a type constructor class. Haskell for the unit is *return* and $(\gg\!=)$ (pronounced "bind") is Haskell's infix notation for the Kleisli extension operation.

```
{-
-- defined in Prelude
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
-}
```

For convenience, we define Kleisli composition.

```
(<.) :: Monad m => (b -> m c) -> (a -> m b) -> a -> m c
g <. f = \ a -> f a >>= g
```

To model the empty context, we define a datatype without data constructors:

```
data Void

instance Show Void
```

and to model extending a given context by an extra variable, we use the Prelude-defined parameterized datatype *Maybe*:

```
{-
-- defined in Prelude
data Maybe a = Nothing | Just a deriving Show

maybe :: c -> (a -> c) -> Maybe a -> c
maybe n j Nothing  = n
maybe n j (Just a) = j a
-}
```

Since we will later want functions on finite contexts (such as substitution rules) to be showable, we define listability, make it clear that finite contexts are listable and define a way for showing a function with listable domain. Of course the idea is to show the graph.

```
class Listable a where
  list :: [a]

instance Listable Void where
  list = []

instance Listable a => Listable (Maybe a) where
  list = Nothing : map Just list

instance (Show a, Listable a, Show b) => Show (a -> b) where
   show f = show (map (\ a -> (a, f a)) list)
```

We record the fact that the *Maybe* functor distributes over any monad:

```
lift :: Monad m => (a -> m b) -> Maybe a -> m (Maybe b)
lift f Nothing  = return Nothing
lift f (Just x) = f x >>= return . Just
```

These preparations made, the untyped λ-calculus syntax and its substitution structure is implemented straightforwardly by one (recursive) parameterized datatype definition and one monad instance declaration as follows. Note that in the definition of $(\gg\!=)$ for abstractions we need *lift*.

```
data Lam a = Var a
           | App (Lam a) (Lam a) | Abs (Lam (Maybe a))
                                              deriving Show
instance Monad Lam where
  return a = Var a
  Var a   >>= g = g a
  App t u >>= g = App (t >>= g) (u >>= g)
  Abs t   >>= g = Abs (t >>= lift g)
```

This is of course with the reservation that, since the model of Haskell is **CPO** and not **Set**, we get lazy rather than well-founded λ-terms.

The code we have shown so far is not new at all and appears already in [7]. But now we can accomplish considerably more. Implementing the syntax of λ-calculus with explicit substitutions following the categorical account via the coend formula is easy in versions of Haskell with the rank 2 type signatures feature (such as the '-98' mode of Hugs or the '-fglasgow-exts' mode of GHC).

```
data LamX b = VarX b
            | AppX (LamX b) (LamX b) | AbsX (LamX (Maybe b))
            | forall a . (Show a, Listable a)
                              => SubX (LamX a) (a -> LamX b)

instance Show b => Show (Lam X b) where ...

instance Monad LamX where
  return a = VarX a
  VarX a   >>= g = g a
  AppX t u >>= g = AppX (t >>= g) (u >>= g)
  AbsX t   >>= g = AbsX (t >>= lift g)
  SubX t f >>= g = SubX t (g <. f)
```

Here it is important to know that in Haskell *forall* in front of a data constructor means existential quantification (in the position of the type of an argument to a data constructor it would mean universal quantification as expected). The motivation for this choice of notation by the Haskell language designers is that a type such as $(\exists a.\ (\mathit{LamX}\ a, a \to \mathit{LamX}\ b)) \to \mathit{LamX}\ b$ (the intended type of SubX) is isomorphic to $\forall a.\ \mathit{LamX}\ a \to (a \to \mathit{LamX}\ b) \to \mathit{LamX}\ b$. The constraint of listability on the bound type variable a of the existential quantifier is necessary to obtain showability of LamX types, but also corresponds nicely to the natural and non-restrictive finiteness constraint on base contexts of explicit substitutions discussed in Section 5. The data constructors for variables, applications and abstractions need new names and redeclaration because Haskell does not support data constructor subtyping and data constructor overloading. The show functions for LamX types have to be defined manually for the mundane reason that Haskell cannot derive them for datatypes with existentially typed components.

We can define the natural embedding of λ-terms to λ-terms with explicit substitutions and the implementation of the evaluation of λ-terms with explicit substitutions to λ-terms. Of course the latter is a post-inverse of the former.

```
embed :: Lam a -> LamX a
embed (Var a)   = VarX a
embed (App t u) = AppX (embed t) (embed u)
embed (Abs t)   = AbsX (embed t)

resolve :: LamX a -> Lam a
resolve (VarX a)   = Var a
resolve (AppX t u) = App (resolve t) (resolve u)
resolve (AbsX t)   = Abs (resolve t)
```

```
resolve (SubX t f) = resolve t >>= resolve . f
```

Now we can proceed to an application. We will implement Krivine's machine for weak head normalization of a λ-term in a lazy fashion. The machine has as states triples (t, g, us) where t is a λ-term with explicit substitutions, g is a substitution of λ-terms with explicit substitutions for the variables of t (environment) and us is a list of λ-terms with explicit substitutions (stack). The intuition is that term t under substitution rule g is the candidate head term for a whnf and list us is the corresponding tail. The state space is Haskell-implemented by a datatype definition where again there is a *forall* meaning existential quantification.

```
data State b = forall a . (Show a, Listable a)
                  => State (LamX a) (a -> LamX b) [LamX b]

instance Show b => Show (State b) where
  show (State t g us) = "Term: " ++ show t ++ "\nSubst: "
                 ++ show g ++ "\nStack: " ++ show us ++ "\n"
```

The transition relation and termination condition of the machine are defined as follows.

```
trans :: (Show b, Listable b) => State b -> State b
trans (State (VarX a)   g us) = State (g a) return  us
trans (State (AppX t u) g us) = State t g ((SubX u g) : us)
trans (State (AbsX t)   g (u : us))
                              = State t (maybe u g) us
trans (State (SubX t f) g us) = State t (g <. f)    us

final :: State b -> Bool
final (State (VarX a) g _ ) = case g a of
                                  VarX _   -> True
                                  SubX _ _ -> False
final (State (AbsX _) _ []) = True
final _ = False
```

We see that the key function *trans* inspects each term and takes appropriate action. Variables are looked up in the environment before reduction continues; in an application, the second term is pushed onto the stack with its environment and reduction continues on the first term; one reduces an abstraction by reducing the body in an extended environment; and one reduces an explicit substitution by composing the environments.

From a given initial state, we can compute the corresponding final state, silently or printing each state passed through.

```
run :: (Show b, Listable b) => State b -> State b
run s = if final s then s else run (trans s)

runIO :: (Show b, Listable b) => State b -> IO (State b)
runIO s = do {print s;
              if final s then return s else runIO (trans s)}
```

To use the machine for wh normalization, we have to initialize it, run it and collect the result.

```
initialize :: (Show b, Listable b) => LamX b -> State b
initialize t = State t return []

collect :: State b -> LamX b
collect (State (VarX a) g us) = app (g a) us
                  where app t [] = t
                        app t (u : us) = app (AppX t u) us
collect (State (AbsX t) g []) = SubX (AbsX t) g
```

Bringing a λ-term with or without explicit substitutions to a whnf is accordingly implementable as follows.

```
reduceX :: (Show b, Listable b) => LamX b -> LamX b
reduceX  = collect . run . initialize

reduce :: (Show b, Listable b) => Lam b -> Lam b
reduce = resolve . reduceX . embed

reduceXIO :: (Show b, Listable b) => LamX b -> IO (LamX b)
reduceXIO t = do {s <- runIO (initialize t); putStr "WhnfX: ";
                  let t' = collect s; print t'; return t'}

reduceIO :: (Show b, Listable b) => Lam b -> IO (Lam b)
reduceIO t = do {t' <- reduceXIO (embed t); putStr "Whnf: ";
                 let t'' = resolve t'; print t''; return t''}
```

Other similar reduction machines can be obtained by modifying the above code. We refer to just one such machine—the eager machine with markers on the stack à la Curien. This has as states triples (t, g, us) or (t, us) where t and g are as before and us is a stack with each entry marked *Left* or *Right* to indicate whether the term in the focus is the function or the argument of an application. To obtain an implementation, it suffices to redefine *State*, *trans*, *final* and *initialize*, *collect* as follows below, leaving the rest of the code intact.

```
data State b = forall a . (Show a, Listable a)
                        => State (LamX a) (a -> LamX b)
                                 [Either (LamX b) (LamX b)]
             | State0 (LamX b) [Either (LamX b) (LamX b)]

instance Show b => Show (State b) where
  show (State t g us) = "Term: " ++ show t ++ "\nSubst: "
                 ++ show g ++ "\nStack: " ++ show us ++ "\n"
  show (State0 t  us) = "Term: " ++ show t ++ "\nStack: "
                 ++ show us ++ "\n"

trans (State (VarX a)   g us) = State0 (g a) us
trans (State (AppX t u) g us) = State t g (Left (SubX u g) : us)
trans (State (AbsX t)   g us) = State0 (SubX (AbsX t) g) us
trans (State (SubX t f) g us) = State t (g <. f) us
trans (State0 t (Left  (SubX u f) : us))
                              = State u f (Right t : us)
trans (State0 t (Right (SubX (AbsX u) f) : us))
                              = State u (maybe t f) us

final :: State b -> Bool
final (State0 _ []) = True
final (State0 _ (Right u : _)) = case u of
                                   SubX (AbsX _) _ -> False
                                   _ -> True
final _ = False

initialize :: (Show b, Listable b) => LamX b -> State b
initialize t = State t return []

collect :: State b -> LamX b
collect (State0 t us) = app t us
             where app t [] = t
                   app t (Left  u : us) = app (AppX t u) us
                   app t (Right u : us) = app (AppX u t) us
```

To summarise, the use of higher-order initial algebras to model explicit substitutions is more than a theoretical exercise. Under an appropriate typing discipline, implementing explicit substitutions in Haskell is no harder than implementing the untyped λ-calculus à la [7, 8]. The resulting code is high-level and modular, follows very closely the motivating mathematical account and is hence easy to read and reason about.

7. Non-Modularity of Higher-Order Syntax

We mentioned in Section 3 that if F and G are 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 language T_{F+G} to a model (a set carrying variables and substitution) is completely determined by its restrictions to two smaller languages T_F and T_G. Intuitively, the reason is that any $(F + G)$-term over X is constructible in a well-founded 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 [21, 20].

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 a first-order language with explicit substitutions amounts to taking its coproduct with the language of 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 [18].

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 bodies. It is not possible to explicitly substitute into a term involving F-operators, i.e., a subterm of the form $sub(\theta, t)$ is illegal, if t involves F-operators. In contrast, E_F allows arbitrary mixes of explicit substitutions and F-operators 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 whether the new variable or the old ones will be used. As a result, the body of an abstraction can only be an application if the application merely 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 body 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-operators. Knowing only the restrictions of f to T_F and E and 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 two languages, the terms of the combined language must be constructible from terms of each given language using the proper substitution of the combined language which obeys the substitution laws. In combinations like the λ-calculus syntax or a first-order language extended with explicit substitutions, the combined language is generated from the given ones by means of a naive substitution.

8. Conclusions and Future Work

Research on higher-order syntax has the potential to produce exciting developments in language design. This paper has taken explicit substitutions, which at first sight may appear to have been simply invented to facilitate the reduction of λ-terms, and shown them to possess fundamental mathematical structure as witnessed by the fact that it only takes very classical categorical machinery, such as coends, to explicate it. We applied this research to present a high-level implementation of reduction machines which, by taking advantage of Haskell's rank-2 signatures, allowed us to avoid commitment to a specific notion of variables bound in an explicit substitution. The resulting code is therefore clearer, more concise, better structured and more manifestly correct than would otherwise be possible. Notice also that there are useful structured recursion schemes for higher-order inductive types (known in functional programming as nested datatypes) [9, 2].

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 Section 7, 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 abstract syntax. 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, related questions arise for the algebras of the functor and representing monad.

The progress we have made so far on these questions has made it apparent that they are intimately related. In particular, the modularity question must be attacked guided by the model theory.

Acknowledgements

We are grateful to our anonymous referees for their useful comments.

N. Ghani's research was partially supported by EPSRC under grant No. GR/M96230/01. T. Uustalu received support from the Estonian Science Foundation under grant No. 5567. Both authors acknowledge also the support from the Royal Society.

References

1. Abadi, M., Cardelli, L., Curien, P.-L. and Levy, J.-J. Explicit substitutions. *J. of Funct. Program.*, **1**(4) (1991), pp. 375-416.
2. Abel, A., Matthes, R. and Uustalu, T. Iteration schemes for higher-order and nested datatypes. *Theor. Comput. Sci.*, **333**(1–2) (2005), pp. 3–66.
3. Aczel, P., Adámek, J., Milius, S. and Velebil, J. Infinite trees and completely iterative theories: A coalgebraic view. *Theor. Comput. Sci.*, **300**(1–3) (2003), pp. 1–45.
4. Adámek, J., Milius, S. and Velebil, J. Free iterative theories: A coalgebraic view. *Math. Struct. in Comput. Sci.*, **13**(2) (2003), pp. 259–320.
5. Adámek, J., Milius, S. and Velebil, J. On rational monads and free iterative theories. In *Proc. of 9th Conf. on Category Theory and Comput. Sci., CTCS'02*, R. Blute and P. Selinger (Eds.), v. 69 of *Electr. Notes in Theor. Comput. Sci.*, Elsevier, 2003.
6. Adámek, J. and Rosický, J. *Locally Presentable and Accessible Categories*, v. 189 of *London Math. Soc. Lecture Note Series*. Cambridge Univ. Press, 1994.
7. Altenkirch, T. and Reus, B. Monadic presentations of lambda terms using generalized inductive types. In *Proc. of 13th Int. Wksh. on Comput. Sci. Logic, CSL'99*, J. Flum and M. Rodríguez-Artalejo (Eds.), v. 1683 of *Lect. Notes in Comput. Sci.*, Springer-Verlag, 1999, pp. 453–468.
8. Bird, R. and Paterson, R. De Bruijn notation as a nested datatype. *J. of Funct. Program.*, **9**(1) (1999), pp. 77–91.
9. Bird, R. and Paterson, R. Generalized folds for nested datatypes. *Formal Aspects of Comput.*, **11**(2) (1999), pp. 200–222.
10. Curien, P.-L. An abstract framework for environment machines. *Theor. Comput. Sci.*, **82**(2) (1991), pp. 389-402.
11. Dubuc, E. J. and Kelly, G. M. A presentation of topoi as algebraic relative to categories or graphs. *J. of Algebra*, **83** (1983), pp. 420–433.

12. Fiore, M. 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*, ACM Press, 2002, pp. 26–37.
13. Fiore, M., Plotkin, G. D. and Turi, D. Abstract syntax and variable binding (extended abstract). In *Proc. of 14th Ann. IEEE Symp. on Logic in Comput. Sci., LICS'99*, IEEE CS Press, 1999, pp. 193–202.
14. Fiore, M. and Turi, D. Semantics of name and value passing. In *Proc. of 16th Ann. IEEE Symp. on Logic in Comput. Sci., LICS'01*, IEEE CS Press, 2001, pp. 93–104.
15. Ghani, N., Lüth, C. and de Marchi, F. Coalgebraic monads. In *Proc. of 5th Wksh. on Coalgebraic Methods in Comput. Sci., CMCS'02*, L. S. Moss (Ed.), v. 65(1) of *Electr. Notes in Theor. Comput. Sci.*, Elsevier, 2002.
16. Ghani, N., Lüth, C., de Marchi, F. and Power, J. Dualising initial algebras. *Math. Struct. in Comput. Sci.*, **13**(2) (2003), pp. 349–370.
17. Ghani, N. and Uustalu, T. Explicit substitutions and higher-order syntax (extended abstract). In *Proc. of 2nd ACM SIGPLAN Wksh. on Mechanized Reasoning about Languages with Variable Binding, MERλIN 2003*, F. Honsell, M. Miculan, A. Momigliano (Eds.), ACM Press, 2003.
18. Hyland, M., Plotkin, G. and Power, J. Combining computational effects: Commutativity and sum. In *Proc. of IFIP 17th World Computer Congress, TC1 Stream / 2nd IFIP Int. Conf. on Theor. Comput. Sci., TCS 2002*, A. Baeza-Yates, U. Montanari, and N. Santoro (Eds.), v. 223 of *IFIP Conf. Proc.*, Kluwer Acad. Publishers, 2002, pp. 474–484.
19. Lüth, C. and Ghani, N. Monads and modular term rewriting. In *Proc. of 7th Int. Conf. on Category Theory and Comput. Sci., CTCS'97*, E. Moggi and G. Rosolini (Eds.), v. 1290 of *Lect. Notes in Comput. Sci.*, Springer-Verlag, 1997, pp. 69–86.
20. Lüth, C. and Ghani, N. Composing monads using coproducts. In *Proc. of 7th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'02*, ACM Press, New York, 2002, pp. 133–144.
21. Lüth, C. and Ghani, N. Monads and modularity. In *Proc. of 4th Int. Wksh. on Frontiers of Combining Systems, FroCoS 2002*, A. Armando (Ed.), v. 2309 of *Lect. Notes in Artif. Intell.*, Springer-Verlag, 2002, pp. 18–32.
22. Mac Lane, S. *Categories for the Working Mathematician*, v. 5 of *Graduate Texts in Math.* Springer-Verlag, 2nd ed., 1997. (1st ed., 1971).
23. Matthes, R. and Uustalu, T. Substitution in non-wellfounded syntax with variable binding. *Theor. Comput. Sci.*, **327**(1–2) (2004), pp. 155–174.
24. McCracken, N. J. The typechecking of programs with implicit type structure. In *Proc. of Int. Symp. on the Semantics of Data Types*,G. Kahn, D. B. MacQueen and G. Plotkin (Eds.), v. 173 of *Lect. Notes in Comput. Sci.*, Springer-Verlag, 1984, pp. 301–315.
25. Miculan, M. and Scagnetto, I. A framework for typed HOAS and semantics. In *Proc. of 5th Int. ACM SIGPLAN Conf. on Principles and Practice of Declarative Programming, PPDP'03*, ACM Press, 2003, pp. 184–194.
26. Milius, S. On iteratable endofunctors. In *Proc. of 9th Conf. on Category Theory and Comput. Sci., CTCS'02*, R. Blute and P. Selinger (Eds.), v. 69 of *Electr. Notes in Theor. Comput. Sci.*, Elsevier, 2003.
27. Moss, L. S. Parametric corecursion. *Theor. Comput. Sci.*, **260**(1–2) (2001), pp. 139–163
28. Peyton Jones, S. (Ed.) Haskell 98 language and libraries: The revised report. *J. of Funct. Program.*, **13**(1) (2003), pp. 1–277.