Syntax and Semantics for Operations with Scopes

Maciej Piróg
University of Wrocław
Poland
maciej.pirog@cs.uni.wroc.pl

Tom Schrijvers
KU Leuven
Belgium
tom.schrijvers@cs.kuleuven.be

Nicolas Wu
University of Bristol
UK
nicolas.wu@bristol.ac.uk

Mauro Jaskelioff
CIFASIS-CONICET
Universidad Nacional de Rosario
Argentina
jaskelioff@cifasis-conicet.gov.ar

Abstract

Motivated by the problem of separating syntax from semantics in programming with algebraic effects and handlers, we propose a categorical model of abstract syntax with so-called *scoped operations*. As a building block of a term, a scoped operation is not merely a node in a tree, as it can also encompass a whole part of the term (a scope). Some examples from the area of programming are given by the operation **catch** for handling exceptions, in which the part in the scope is the code that may raise an exception, or the operation **once**, which selects a single solution from a nondeterministic computation. A distinctive feature of such operations is their behaviour under program composition, that is, syntactic substitution.

Our model is based on what Ghani *et al.* call the monad of explicit substitutions, defined using the initial-algebra semantics in the category of endofunctors. We also introduce a new kind of multi-sorted algebras, called *scoped algebras*, which serve as interpretations of syntax with scopes. In generality, scoped algebras are given in the style of the presheaf formalisation of syntax with binders of Fiore *et al.* As the main technical result, we show that our monad indeed arises from free objects in the category of scoped algebras.

Importantly, we show that our results are immediately applicable. In particular, we show a Haskell implementation together with practical, real-life examples.

1 Introduction

1.1 The Big Picture: Formalised Abstract Syntax

This paper is about formal models of abstract syntax and their connection with semantics, mainly in the context of programming languages. The simplest model of abstract syntax is given by the set of terms over a signature, parametrised by the set of variables. Terms can be generalised to free monads generated by an endofunctor, which allows one to capture more intricate kinds of signatures, and move to different categories, possibly more amenable for modelling and reasoning about programming languages. However, to model syntax of programming calculi, one needs more advanced structures; for example, Fiore *et al.* [10] model syntax with binders as algebras on certain presheaf categories.

There are two major applications of constructing formal models of syntax. First, they often lead to implementations, either in proof assistants to reason about programming calculi (for example, the syntax with binders of Fiore *et al.* is implementable using nat-indexed types, see [1]), or in programming languages themselves to achieve metalinguistic abstraction (see [26, 27]).

The other application is in investigating the connection between syntax and semantics. The object that represents syntax is often given by a monad in which the monadic structure captures substitution. Such a monad often comes about from a free–forgetful adjunction, therefore validating the syntax as arising from free objects in a wider category of semantic objects. For example, as described in more detail in Section 2, the (algebraically) free monad is given by free objects in the category of algebras for an endofunctor.

In this paper, we introduce a model of syntax with *scoped operations*; see Wu *et al.* [29]. We show both an implementation in Haskell (Section 6) and an appropriate adjunction (Section 4).

1.2 Motivation: Non-algebraic Operations

Since the introduction of the computational λ-calculus by Moggi [20], monads have become the standard abstraction to capture different notions of computational effects, such as nondeterminism, exceptions, mutable state, concurrency, and so on. Each strong monad M on a Cartesian closed category is equipped with the *bind* operator, which can be given a higher-order type $\gg\!= : MA \times (A \to MB) \to MB$. Intuitively, it allows for the sequential composition of two computations, in which the second computation is parametrised by the result of the first one.

While in general monads provide an interface for composing existing computations and creating pure computations, they abstract away the constructs that actually create effectful computations. It has been a goal of the research programme initiated by Plotkin and Power (see [24]) to better understand such constructs in terms of *operations*. As our running example, we consider a simple version of nondeterminism. If a monad M implements nondeterminism, we assume it to have three operations: binary $\mathbf{or} : MA \times MA \to MA$ (the intuitive meaning of $\mathbf{or}(p_1, p_2)$ is nondeterministic choice between programs p_1 and p_2), nullary $\mathbf{fail} : 1 \to MA$ (intuitively, a computation that gives no results), and unary $\mathbf{once} : MA \to MA$ (intuitively, select the first available result from the computation in the argument). Note that we do not assume commutativity of the choice, that is, in general $\mathbf{or}(a, b) \neq \mathbf{or}(b, a)$ when $a \neq b$. One important observation by Plotkin and Power [23] is that such operations can be categorised into two different kinds: *algebraic* and *non-algebraic* ones. The defining property of the former is that they commute with bind. That is, an n-ary operation $\mathbf{op} : (MA)^n \to MA$ is algebraic if the following equality holds:

$$\mathbf{op}(x_1, \ldots, x_n) \gg\!= k \;=\; \mathbf{op}(x_1 \gg\!= k, \ldots, x_n \gg\!= k) \qquad (1)$$

PL'17, January 01–03, 2017, New York, NY, USA
2017.

In the case of nondeterminism, **or** and **fail** are algebraic. For example, one expects the following equality (for brevity, *return* injections of pure values into the monad are implicit):

$$\mathbf{or}(1,5) \gg\!= k \;=\; \mathbf{or}(k\,1, k\,5)$$

Indeed, choosing between 1 and 5, and then continuing with the computation k parameterised by the result is the same as choosing between the computation k parameterised by 1 and the computation k parameterised by 5. On the other hand, **once** is not algebraic. Following the intuitive semantics, $\mathbf{once}(\mathbf{or}(1,5)) = 1$, and therefore we have the following:

$$\mathbf{once}(\mathbf{or}(1,5)) \gg\!= \lambda x.\, \mathbf{or}(x, x{+}1) \;=\; 1 \gg\!= \lambda x.\, \mathbf{or}(x, x{+}1) \;=\; \mathbf{or}(1,2)$$

If **once** were an algebraic operation, it would commute with bind, and we would obtain a different result:

$$\mathbf{once}(\mathbf{or}(1,5)) \gg\!= \lambda x.\, \mathbf{or}(x, x{+}1) \;=\; \mathbf{once}(\mathbf{or}(\mathbf{or}(1,2), \mathbf{or}(5,6))) \;=\; 1$$

The above is not what one expects and hence **once** is not algebraic. The recognition of the class of algebraic operations was a stepping stone, as they arise as operations coming from algebraic presentations of monads, hence allowing for equational reasoning about effectful programs [22] (see also [14]).

Later, Plotkin and Pretnar [25] explained a broad spectrum of non-algebraic operations in terms of *handlers*. In their setting, algebraic operations are purely syntactic constructs, with no semantics associated *a priori*. A number of algebraic operations put together form a signature, which then induces syntax, which is given by the set of terms over the signature, or, more generally, a free monad, in which bind is given by substitution for variables. A handler gives semantics to computations built from algebraic operations: it specifies a carrier and gives an interpretation to the algebraic operations. The combination of algebraic operations and handlers has sparked a new phase of practical adoption, with a bevy of new programming language and library designs [4, 6, 8, 17, 18]. Formally, handlers arise from Eilenberg–Moore algebras for the free monad induced by the signature.

Looking at our running example through this lens, the operations **or** and **fail** are algebraic, hence they have no semantics by themselves. One possible implementation of **once** as a handler uses the same free monad as its carrier, and is obtained by preserving pure values and folding the syntax with the following recursive equations:

$$\begin{aligned} &\mathit{once}(\mathbf{fail}) = \mathbf{fail} \\ &\mathit{once}(\mathbf{or}(p_1, p_2)) = \mathit{once}(p_1) && \text{(if } \mathit{once}(p_1) \neq \mathbf{fail}\text{)} \\ &\mathit{once}(\mathbf{or}(p_1, p_2)) = \mathit{once}(p_2) && \text{(otherwise)} \end{aligned}$$

A shortcoming of this approach is that handlers play two simultaneous roles: they create a scope that delimits a part of the syntax to be interpreted, and at the same time they interpret the syntax in that scope. That is, we can see a computation constructed solely with algebraic operations as a syntactic being, and we defer giving it semantics until it is put in an appropriate context (that is, a handler), while non-algebraic operations (given by handlers) always come together with an interpretation. Therefore, by using handlers to model operations that delimit a scope of other computations, programs are a mix of syntax and semantics. This is problematic, because it forbids certain interpretations of programs [29]. Is it possible to have such *scoped operations* as syntax, thus keeping a clean separation of syntax and semantics?

In this paper, we show how to achieve this by expressing the abstract syntax as an initial algebra in the category of endofunctors on a given base category, in the style of Ghani *et al.* [13]. We also identify a category of semantic objects that give a way of interpreting our syntax. In particular, we can construct computations using **or**, **fail**, and **once** without committing to any semantics of these operations. Later on, we can interpret them using any chosen semantics (for example, accumulating results on a list or using a random choice).

1.3 Overview

We tackle the problem of formalising abstract syntax with scopes from two different ends. First, in Section 3, we show a monad E that allows for scoped operations by keeping an explicit continuation, which is parametrised by the results of the syntax in the scope. The monad E is a generalisation of Ghani *et al.*'s [13] monad of explicit substitutions, defined using initial algebras over the category of endofunctors $\mathscr{C}^{\mathscr{C}}$ for a base category $\mathscr{C}$.

Unfortunately, the monad E does not come equipped with a natural notion of an interpretation of syntax, as the obvious choice, Eilenberg–Moore algebras, turns out to be inadequate for this purpose. We fix this in Section 4, in which we tackle the problem from the other end. We first define interpretations of syntax with scoped operations as certain algebras over the category of $\mathbb{N}$-indexed objects $\mathscr{C}^{|\mathbb{N}|}$, in the style of Fiore *et al.*'s [10] presheaf formalisation of syntax with binders. These algebras induce a free monad M on $\mathscr{C}^{|\mathbb{N}|}$, which can be projected on the base category $\mathscr{C}$, giving a new monad $\lfloor M \rceil$ that models syntax naturally interpreted by our algebras.

In Section 5, as our main technical result, we construct an isomorphism between the monads E and $\lfloor M \rceil$. This means that the intuitive ideas for how syntax and semantics for scoped operations should work coincide. The main technical obstacle is that while the monad E is defined as a carrier of an initial algebra (in $\mathscr{C}^{\mathscr{C}}$), the monad $\lfloor M \rceil$ is expressed using a composition of adjunctions, hence the isomorphism does not arise directly from the universal property of E or M. We solve this by lifting the monad E to the category $\mathscr{C}^{|\mathbb{N}|}$.

Both our monads are variations on the theme of algebraic operations that represent opening and closing brackets. Brackets were already briefly considered by Wu *et al.* [29], but dismissed on the grounds that, in the usual approach to syntax, there is no way to guarantee that the brackets are well-paired. Here, we avoid this problem either by encoding the continuation as an explicit substitution in the monad E, or by employing types to keep the structural properties in $\lfloor M \rceil$.

Our constructions are amenable to be directly encoded in a language with a rich type system. In Section 6, we show a Haskell implementation and demonstrate it in Section 7 on two examples, state with local variables and resumption-based concurrency.

1.4 Contributions

The original contributions of this paper can be summarised as follows:

- We observe a relationship between scoped operations and explicit substitutions in the sense of Ghani *et al.* [13]. We use this observation to define the monad E that models abstract syntax with scopes.

- We introduce *scoped algebras*, which serve as interpretations of syntax with scopes. We show that the monad that arises from free scoped algebras, M, leads to a monad on the base category, $\lfloor M \rceil$, which is naturally interpreted by scoped algebras.
- We show that the monads E and $\lfloor M \rceil$ are isomorphic, which means that the two different approaches to scoped syntax are actually equivalent.
- This equivalence enables us to give a compact implementation in Haskell. We use it to show that our framework is expressive enough to capture a number of examples of effects in which scoped operations play a vital role: nondeterminism (with the scoped operation **once**), exceptions (with **catch**), state with local variables (with **local**), or resumption-based concurrency (with **fork** and **atomic**).

2 Background: Adjoint-theoretic Approach to Syntax and Semantics

Formalising various kinds of syntax can be discussed at many levels of generality. Hence, we show how one can formalise the usual story about syntax with algebraic operations and its interpretation at the level of generality that we aim at, that is, when the signature is given by an endofunctor on a category with rich enough structure, and syntax is given by the free monad over that signature.

As for the notation, we write $\mathscr{A}, \mathscr{C}$ to denote categories, while Σ and Γ stand for endofunctors that represent signatures. We denote natural transformations using the applicative notation, for example, $\alpha A : \Sigma A \to \Gamma A$. When the type is given, we often drop the component, and write simply $\alpha : \Sigma A \to \Gamma A$. In abuse of notation, monads are identified with their underlying endofunctors, and the monadic structure is always denoted as η (unit) and μ (multiplication).

First, assume that we work in a category $\mathscr{A}$ with finite coproducts. These include the initial object, denoted 0. Given an endofunctor Σ, which represents a signature, a Σ-*algebra* is a tuple $\langle A,\ a : \Sigma A \to A\rangle$, where A (the *carrier*) is an object in $\mathscr{A}$. Such algebras form a category, denoted Σ-**Alg**, with morphisms between $\langle A, a\rangle$ and $\langle B, b\rangle$ given by morphisms $f : A \to B$ in $\mathscr{A}$ such that $b \cdot \Sigma f = f \cdot a$. If the initial algebra for an endofunctor Σ exists, we denote its carrier $\mu\Sigma$ or $\mu X.\, \Sigma X$.

Assume that for all objects A in $\mathscr{A}$, the initial algebra with carrier $\Sigma^* A = \mu X.(\Sigma X + A)$ exists. This family of initial algebras form a monad Σ^*. It is called the *(algebraically) free monad* [3] generated by Σ, and it represents syntax over the signature Σ. This monad arises from an important adjunction, which explains how one can define interpretations of terms. First, we consider the obvious *forgetful* functor $U : \Sigma\text{-}\mathbf{Alg} \to \mathscr{A}$ defined as $U\langle A, a\rangle = A$. It has a left adjoint, the *free* functor $FA = \langle \Sigma^* A,\ \mathrm{cons}A : \Sigma\Sigma^* A \to \Sigma^* A\rangle$. Then, the monad Σ^* is given as the composition UF. (The family $\mathrm{cons}A$ is natural in A, hence we use the notation for natural transformations.) The situation is summarised in the following diagram:

$$UF = \Sigma^* \circlearrowright \mathscr{A} \underset{U}{\overset{F}{\underset{\longleftarrow}{\overset{\longrightarrow}{\bot}}}} \Sigma\text{-}\mathbf{Alg} \tag{2}$$

The notion of interpretation arises from the same adjunction. Consider the associated natural isomorphism:

$$\Phi(A, \langle B, b\rangle) : \mathrm{Hom}_{\mathscr{A}}(A, B) \to \mathrm{Hom}_{\Sigma\text{-}\mathbf{Alg}}(\langle \Sigma^* A, \mathrm{cons}A\rangle, \langle B, b\rangle).$$

Given a value of the type $\Sigma^* A$ (understood intuitively as a Σ-term with variables from the set A), we can interpret it using a carrier B by providing a morphism $f : A \to B$ to interpret the variables, and an algebra $b : \Sigma B \to B$ to interpret operations. The morphism that interprets the entire 'term' is then given by $U(\Phi(A, \langle B, b\rangle) f) : \Sigma^* A \to B$. Intuitively, it applies f to the variables and then folds the structure using b.

An important property of the adjunction $F \dashv U$ is that it is *strictly monadic*. This means that the category Σ-**Alg** is isomorphic to the Eilenberg–Moore category of the monad Σ^* (that is, the Σ^*-algebras $\langle A, a\rangle$ for which $a \cdot \eta = \mathrm{id}$ and $a \cdot \mu = a \cdot \Sigma^* a$), and the adjunction $F \dashv U$ is essentially the Eilenberg–Moore adjunction of Σ^*. This fact is useful for technical purposes, but it also gives us an intuitive understanding of the category of Σ-algebras as the category of interpretations (models) of syntax.

3 Scopes via Explicit Substitutions

In this section, we introduce a monad E that formalises syntax with both algebraic operations and scoped operations. Our monad is a generalisation of Ghani *et al.*'s [13] monad of explicit substitutions. In order to motivate this construction, we first discuss it informally.

As our running example, consider two algebraic operations for nondeterminism: binary **or** and nullary **fail**. Additionally, consider a unary operation **once**. These operations just define syntax, but the intended meaning is that **or** nondeterministically chooses a computation, **fail** is a computation that gives no solutions, and **once**, just as a Prolog operation with the same name, chooses the first solution from the computation in its argument.

One way to approach the syntax generated by these operations is to appropriately extend the notion of terms and substitution. That is, we still consider tree-like structures, in which nodes correspond to operations, while leaves correspond to variables, understood as values returned by the nondeterministic computation. The nodes representing algebraic operations behave exactly as in the case of terms, that is, **or** is a node with two children, and **fail** is a node with zero children. For such nodes, the monadic bind operator behaves like the usual substitution, that is, it is defined recursively as in the equation (1). However, as illustrated in the introduction, we cannot simply substitute in the argument of **once**. This is because in the expression $\mathbf{once}(t) \gg\!= k$, the computation t is in the scope of **once**, but the results of applying k are not. Thus, since we cannot perform the usual substitution, we replace it by an *explicit substitution*: each node representing **once** is actually a pair consisting of the computation in the scope (t) and a continuation (k). More explicitly, the computation $\mathbf{once}(\mathbf{or}(1, 5))$ is represented by the expression $\mathbf{or}(1, 5)$ paired with the continuation given by the unit η of the monad E, which, as in the case of the free monad, embeds variables as terms. Then, the bind operator $\gg\!= k$ recursively Kleisli-composes k with the continuation. In particular, the expression $\mathbf{once}(\mathbf{or}(1, 5)) \gg\!= k$ represents the following syntax tree, where $\circ$ denotes Kleisli composition:

$$\left(\begin{array}{c}\mathbf{once}\\ \swarrow\ \ \searrow\\ \mathbf{or}\qquad \eta\\ \swarrow\ \searrow\\ 1\quad 5\end{array}\right) \gg\!= k \quad = \quad \begin{array}{c}\mathbf{once}\\ \swarrow\ \ \searrow\\ \mathbf{or}\qquad k \circ \eta\\ \swarrow\ \searrow\\ 1\quad 5\end{array} \quad = \quad \begin{array}{c}\mathbf{once}\\ \swarrow\ \ \searrow\\ \mathbf{or}\qquad k\\ \swarrow\ \searrow\\ 1\quad 5\end{array}$$

Note that in general the first child of the node representing **once** has the type EA for a type A, while the type of the second child is

given by the exponential $(EB)^A$ for a type B. Moreover, the type A is not visible from above the **once** node, so A is an existential type.

The construction sketched above is similar to what Ghani *et al.* [13] call the *explicit substitution* monad. What is new is the observation that it can be used to model scoped operations if appropriately generalised, as we now describe.

The reason why we need to generalise the explicit substitution monad is that we may want more than one scoped operation in the signature, and that we sometimes want scoped operations to have more intricate arities, which allow them to carry more data than just one encompassed expression. One example is **catch**, which allows for handling exceptions. It can contain not only an expression that may throw an exception, but also a number of expressions used for handling. Thus, given a fixed set of exceptions S, the **catch** operation has two arguments of the types EA and $(EA)^S$ respectively. To account for this in generality, our monad E is parametrised by two different endofunctors: Σ for representing algebraic operations, and Γ for operations that create a scope. For example, the signatures for nondeterminism can be given as $\Sigma X = X \times X + 1$ and $\Gamma X = X$, while the signatures for exceptions can be given as $\Sigma X = S$ and $\Gamma X = X \times X^S$.

Following Ghani *et al.*, the intuition about the existential type can be formalised using a coend. Simplifying things a bit (and ignoring size issues), given two endofunctors Σ and Γ, we want the monad E over a category $\mathscr{C}$ to behave like a fixed point of the following equation:

$$EA \;\cong\; A \;+\; \Sigma(EA) \;+\; \int^{X \in \mathscr{C}} \Gamma(EX) \;\times\; (EA)^X$$

Note that the coend computes the left Kan extension of ΓE along the identity applied to the object EA. This means that we can rewrite the equation above as follows (for the details, we refer to [13]):

$$EA \;\cong\; A \;+\; \Sigma(EA) \;+\; \Gamma(E(EA))$$

Formally, we consider the category $\mathscr{C}^{\mathscr{C}}$ with endofunctors on $\mathscr{C}$ as objects and natural transformations as morphisms. This category inherits coproducts from $\mathscr{C}$, given as $(G + H)A = GA + HA$. We define an endofunctor $G : \mathscr{C}^{\mathscr{C}} \to \mathscr{C}^{\mathscr{C}}$ with the following mapping of objects:

$$GH = \mathsf{Id} \;+\; \Sigma H \;+\; \Gamma HH$$

The action on morphisms is given by appropriate horizontal compositions. If G has an initial algebra μG, we use its carrier to define the endofunctor part of the monad E:

$$E = \mu G$$

The monadic structure of E is given by substitution in the 'Id' component. We discuss it formally in Section 4.

Sometimes, if we need to be explicit about the involved signatures, we write them as superscripts, i.e., $E^{\Sigma,\Gamma}$. Ghani *et al.*'s [13] monad can be obtained by instantiating Γ to Id, that is, as $E^{\Sigma,\mathsf{Id}}$, while the free monad generated by Σ is given by E^{Σ,K_0}, where K_0 is the constant endofunctor $K_0 X = 0$.

4 Interpreting Syntax with Scopes

Now that we have defined syntax for operations with scopes, we show how to define semantics for such syntax. The most obvious idea is to mimic the free-forgetful adjunction (2), and use the Eilenberg-Moore adjunction of the monad E. However, as we discuss in this section, syntax in scopes does not interact with the monadic structure of E, and so Eilenberg-Moore algebras of E do not necessarily respect the structure within the scopes.

Thus, we propose a solution by looking at the problem from another angle. We *first* define a category of a new kind of algebras, which, we believe, are intuitively the right ones to interpret operations with scopes, and only then we construct an adjunction that plays the role of $F \dashv U$ from (2). This new adjunction gives us a monad on $\mathscr{C}$ that models syntax, which can be interpreted using the new kind of algebras, just as the syntax given by free monads can be interpreted using Σ-algebras. As our main technical result, in Section 5 we show that this monad coincides with E.

4.1 Monadic Structure and Algebras for the Monad E

As noticed by Ghani *et al.* [13], the monadic structure of their monad of explicit substitutions coincides with how the regular substitution in the free monad works. Indeed, one can give a monadic structure to E by showing that it is a free monad on $\mathscr{C}$ (provided, as usual, that enough initial algebras exist). First, define an endofunctor E' on $\mathscr{C}^{\mathscr{C}}$ as follows:

$$E' = \mu H.\ \mathsf{Id} + \Sigma H + \Gamma E H$$

By the 'diagonal rule' [2], one can easily show that E' is isomorphic to E. Moreover, if the free monad $(\Sigma + \Gamma E)^*$ exists, one can see that it is isomorphic to E' as a functor. Hence, we obtain that

$$E \cong (\Sigma + \Gamma E)^*,$$

which gives us a monadic structure for E.

The fact that E is a free monad and the fact that the adjunction $F \dashv U$ is strictly monadic reveal that an Eilenberg-Moore algebra for E can be seen as a triple

$$\langle A,\ a : \Sigma A \to A,\ g : \Gamma E A \to A \rangle,$$

where A is an object in $\mathscr{C}$, while a and g are morphisms in $\mathscr{C}$. Intuitively, such an algebra interprets a term by folding the structure from the leaves and working its way up to the root. It interprets an algebraic operation from the signature Σ exactly as in the case of free monads, while in the case of a scoped operation from Γ, no conditions are imposed on g, so it does not have to respect any structural properties of E. In particular, it does not have to apply itself recursively to the syntax in the scope. Therefore, we cannot accept Eilenberg-Moore algebras for E as the 'right' notion of interpretation of syntax with scopes.

4.2 Scoped Algebras

Intuitively, in the usual approach to syntax, an interpretation of a term can be understood as folding the structure from the leaves and working its way up to the root. In the case of a scoped operation, we first want to fold the continuation, and then use the result as the initial value for folding the part in the scope. However, the algebra that interprets the scope might need a different type of carrier, as we show in the examples further in this section.

Thus, the new kind of algebras, which we call *scoped algebras*, are different from F-algebras in that they have a family of $\mathscr{C}$-objects $A_0, A_1, \ldots$ as carriers, each representing a carrier for a level of nesting of scopes. In a scoped algebra, when entering a scope, one needs to *promote* the current value of the type A_n to the type A_{n+1}, and, dually, one *demotes* a value of the type A_{n+1} to A_n when interpreting the operation that creates the scope.

To formalise this, we work in the category of $\mathbb{N}$-indexed $\mathscr{C}$-objects. Let $|\mathbb{N}|$ be the discrete category of natural numbers, in which objects are given by natural numbers, while the only morphisms are trivial identities. Our category of interest is then $\mathscr{C}^{|\mathbb{N}|}$, that is, the category of functors of the type $|\mathbb{N}| \to \mathscr{C}$ with natural transformations as morphisms. More explicitly, an object in $\mathscr{C}^{|\mathbb{N}|}$ is an $\mathbb{N}$-indexed family of $\mathscr{C}$-objects $A = \{A_n\}_{n\in\mathbb{N}}$. A morphism between A and $B = \{B_n\}_{n\in\mathbb{N}}$ is an $\mathbb{N}$-indexed family of $\mathscr{C}$-morphisms $\{f_n : A_n \to B_n\}_{n\in\mathbb{N}}$. In particular, there are no coherence conditions for the components of morphisms at different indices. Moreover, as a functor category, $\mathscr{C}^{|\mathbb{N}|}$ inherits coproducts from $\mathscr{C}$, defined as $(A + B)_n = A_n + B_n$.

As the next step in our construction, we define two endofunctors on $\mathscr{C}^{|\mathbb{N}|}$. The first one is called *shift left*, denoted $\triangleleft$. It moves every component one notch 'to the left', forgetting the object at index 0:

$$(\triangleleft A)_i = A_{i+1}$$

The other endofunctor, *shift right*, denoted $\triangleright$, moves every component one notch 'to the right', sticking the initial object 0 at index 0:

$$(\triangleright A)_0 = 0 \qquad (\triangleright A)_{i+1} = A_i$$

The actions on morphisms are obvious. Additionally, we can lift every endofunctor Σ on $\mathscr{C}$ to an endofunctor $\overline{\Sigma}$ on $\mathscr{C}^{|\mathbb{N}|}$ by applying it at each index:

$$(\overline{\Sigma}A)_n = \Sigma(A_n)$$

With this, we define scoped algebras as follows:

Definition 4.1. Given two endofunctors Σ and Γ, a *scoped algebra* is a quadruple

$$\langle A,\ a : \overline{\Sigma}A \to A,\ d : \overline{\Gamma}\triangleleft A \to A,\ p : A \to \triangleleft A\rangle,$$

where A is an object in $\mathscr{C}^{|\mathbb{N}|}$, while a, d ('demote'), and p ('promote') are morphisms in $\mathscr{C}^{|\mathbb{N}|}$.

Example 4.2. Assume that the base category is **Set**, and consider the binary operation **or**, nullary **fail**, and scoped unary **once**, as discussed in Section 3. Thus, we obtain $\Sigma X = X \times X + 1$ and $\Gamma X = X$. For readability, we write **or** and **fail** to denote the left and right injections in Σ respectively. The usual interpretation of nondeterminism that collects results on a list can be given as follows. For a set X, the carrier is given as $A_n = \mathsf{List}^{n+1}\,X$, where List^n is the n-fold composition of the list endofunctor. The interpretation of **or** is given as concatenation, that is, $a_n(\mathbf{or}(x, x')) = x +\!\!+\, x'$ for all indices $n \in \mathbb{N}$, while **fail** is interpreted as the empty list: $a_n(\mathbf{fail}()) = []$. The promotion morphism is given as the singleton: $p_n(x) = [x]$. The demotion morphism (that is, the interpretation of **once**) selects the first element if it exists, namely $d_n([x, \ldots]) = x$ and $d_n([]) = []$.

The definition of scoped algebras can be reformulated by noticing the simple fact that the functor $\triangleright$ is a left adjoint to $\triangleleft$. It follows that promotion morphisms $A \to \triangleleft A$ are in a 1-1 correspondence with morphisms $\triangleright A \to A$. Thus, scoped algebras are simply algebras for the endofunctor $\overline{\Sigma} + \overline{\Gamma}\triangleleft + \triangleright : \mathscr{C}^{|\mathbb{N}|} \to \mathscr{C}^{|\mathbb{N}|}$, and therefore objects in the category $(\overline{\Sigma} + \overline{\Gamma}\triangleleft + \triangleright)$-**Alg**.

4.3 Syntax Induced by Scoped Algebras

Scoped algebras are simply algebras for an endofunctor on $\mathscr{C}^{|\mathbb{N}|}$, and therefore when enough initial algebras exist, the forgetful functor $U : (\overline{\Sigma} + \overline{\Gamma}\triangleleft + \triangleright)\text{-}\mathbf{Alg} \to \mathscr{C}^{|\mathbb{N}|}$ has a left adjoint F, and this adjunction gives rise to a free monad M defined as follows:

$$M = UF = (\overline{\Sigma} + \overline{\Gamma}\triangleleft + \triangleright)^* : \mathscr{C}^{|\mathbb{N}|} \to \mathscr{C}^{|\mathbb{N}|}$$

Moreover, we can obtain a monad on $\mathscr{C}$ by noticing the following fact:

Theorem 4.3. *Consider the forgetful functor $\downharpoonleft\, : \mathscr{C}^{|\mathbb{N}|} \to \mathscr{C}$ that projects on $\mathscr{C}$ the value at the first index, that is:*

$$\downharpoonleft A = A_0$$

It has a left adjoint $\upharpoonright\, : \mathscr{C} \to \mathscr{C}^{|\mathbb{N}|}$ given as follows:

$$(\upharpoonright X)_0 = X \qquad (\upharpoonright X)_{n+1} = 0$$

Since adjunctions compose, we obtain an adjunction between $\mathscr{C}$ and $(\overline{\Sigma} + \overline{\Gamma}\triangleleft + \triangleright)$-**Alg**, and, as a consequence, a monad $\downharpoonleft M \upharpoonright\, = \,\downharpoonleft UF \upharpoonright$ on $\mathscr{C}$, as shown in the following diagram:

$$\downharpoonleft M \upharpoonright \circlearrowright \mathscr{C} \underset{\downharpoonleft}{\overset{\upharpoonright}{\rightleftarrows}} \underset{M = UF = (\overline{\Sigma} + \overline{\Gamma}\triangleleft + \triangleright)^*}{\mathscr{C}^{|\mathbb{N}|} \circlearrowleft} \underset{U}{\overset{F}{\rightleftarrows}} (\overline{\Sigma} + \overline{\Gamma}\triangleleft + \triangleright)\text{-}\mathbf{Alg} \tag{3}$$

For a better intuitive understanding of the monad $\downharpoonleft M \upharpoonright$, we first discuss the monad M. Intuitively, for an object A in $\mathscr{C}^{|\mathbb{N}|}$ and an index $n \in \mathbb{N}$, we think of a value of $(MA)_n$ as encoding a (sub)term whose root is in n nested scopes. Specifically, for $n > 0$, the object MA at n is given via Lambek's lemma as follows:

$$\begin{aligned}(MA)_n &\cong (A + \overline{\Sigma}MA + \overline{\Gamma}\triangleleft MA + \triangleright MA)_n\\ &= A_n + \Sigma(MA)_n + \Gamma(MA)_{n+1} + (MA)_{n-1}\end{aligned}$$

This could be read as if there were four constructors used to obtain the values of the type $(MA)_n$. The first one is a variable A_n. The second one is an algebraic operation given by the signature Σ with arguments of the type $(MA)_n$, which means that they are situated in the same number of nested scopes. The third constructor gives us a scoped operation from the signature Γ. This operation creates a new scope (it is an 'opening bracket'), so its arguments are of the type $(MA)_{n+1}$, which means that they are surrounded by $n + 1$ scopes. The last constructor is a 'closing bracket', which means that its argument $(MA)_{n-1}$ is a part of the continuation, so it is outside of the n-th scope, hence it is surrounded by $n - 1$ nested scopes.

For $n = 0$, the following holds:

$$(MA)_0 \cong A_0 + \Sigma(MA)_0 + \Gamma(MA)_1 + 0 \cong A_0 + \Sigma(MA)_0 + \Gamma(MA)_1$$

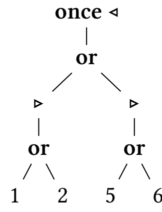
This means that at index 0 ('no surrounding scopes'), we cannot put a closing bracket, which guarantees that every closing bracket matches an opening one.

Thus, we can think of an expression of the shape $\triangleleft M \triangleright$ as M put in brackets that delimit the scope. (Coincidentally, the symbols '$\triangleleft$' and '$\triangleright$' look a bit like opening and closing angle brackets.) Since $(\downharpoonleft A)_n = 0$, for all $n > 0$, by sandwiching the monad M in the adjunction $\upharpoonright \dashv \downharpoonleft$, we ensure that there are no variables at indices $n > 0$, that is, within at least one scope. This is because we substitute only in the outermost continuation; in other words, when we are not in a scope.

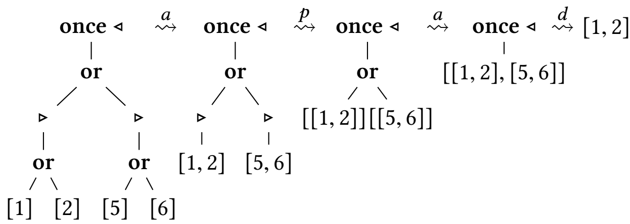
Example 4.4. Consider Example 4.2. The expression

$$\mathbf{once}(\mathbf{or}(\mathit{return}\,1, \mathit{return}\,5)) \gg\!= \lambda x.\, \mathbf{or}(\mathit{return}\,x, \mathit{return}\,(x+1)) \tag{4}$$

can be intuitively understood as the following structure:



If we interpret **or** as accumulating results on a list, and **once** as selecting the first result, the left-hand side of the bind operator in (4) becomes the singleton [1], so the whole expression is interpreted as $[1] \ggg \lambda x.\, \mathbf{or}(\mathit{return}\, x, \mathit{return}\,(x+1))$, which obviously becomes $[1,2]$. Indeed, this is what we obtain by interpreting the variables as singletons and applying the scoped algebra from Example 4.2:



Example 4.5. Let $\mathscr{C} = \mathbf{Set}$. The signature for exceptions coming from a set S can be given as $\Sigma X = S$ and $\Gamma X = X \times X^S$, where, intuitively, the second component of Γ is the code that handles exceptions. We can give these signatures the usual semantics using an algebra with the carrier $A_n = H^{n+1}X$, for a set X, where $HB = B + S$. The associated morphisms are defined as $a_n = \mathsf{inr} : S \to H^n X + S$, together with $p_n = \mathsf{inl} : H^{n+1}X \to H^{n+1}X + S$, and $d_n : (H^{n+1}X + S) \times (H^{n+1}X + S)^S \to H^{n+1}X$ given as:

$$d_n(\mathsf{inl}\; a, f) = a \qquad d_n(\mathsf{inr}\; s, f) = ([\mathsf{id}, \mathsf{inr}] \cdot f)(s)$$

5 Equivalence of the Two Models

At this point, we have two monads on $\mathscr{C}$ that we can use to represent syntax with scopes: the 'explicit substitution' monad E and the 'indexed objects' monad $\downharpoonleft M \upharpoonright$. As it turns out, they are essentially the same monad:

Theorem 5.1. *The monads E and $\downharpoonleft M \upharpoonright$ are isomorphic in the category of monads on $\mathscr{C}$ and monad morphisms.*

We sketch a proof. For all objects A in $\mathscr{C}^{|\mathbb{N}|}$, we use the notation cons to denote the 'constructor' components of the initial algebra $\mathsf{in} = [\eta, \mathsf{cons}, \mathsf{cons}, \mathsf{cons}] : A + \overline{\Sigma} MA + \overline{\Gamma}{\triangleleft} MA + {\triangleright} MA \to MA$. We use the same notation for the components of the initial algebra $\mathsf{in} = [\eta, \mathsf{cons}, \mathsf{cons}] : \mathsf{Id} + \Sigma E + \Gamma EE \to E$. Thus, the notations in and cons are overloaded, but it is always clear which morphism we mean thanks to explicit types. Another overloaded symbol is ϵ, which denotes the counit of an adjunction known from the context.

We define the isomorphism $i : E \to \downharpoonleft M \upharpoonright$. Intuitively, it flattens the nested structure of E and inserts opening and closing brackets to mark the original scope. For this, we first introduce a natural transformation in $\mathscr{C}^{|\mathbb{N}|}$ that can put M-structure into brackets: $M \to {\triangleleft} M {\triangleright}$. It is done using a distributive law $\lambda : M{\triangleleft} \to {\triangleleft} M$ of the monad M over the endofunctor $\triangleleft$. For an object A, it is given as the unique morphism induced by the algebra $[\lambda^{\mathrm{var}}, \lambda^{\Sigma}, \lambda^{\triangleleft}, \lambda^{\triangleright}] : {\triangleleft} A + \overline{\Sigma}{\triangleleft} MA + \overline{\Gamma}{\triangleleft\triangleleft} MA + {\triangleright\triangleleft} MA \to {\triangleleft} MA$, where:

$$\lambda^{\mathrm{var}} = \left({\triangleleft} A \xrightarrow{{\triangleleft}\eta} {\triangleleft} MA\right) \qquad \lambda^{\Sigma} = \left(\overline{\Sigma}{\triangleleft} MA = {\triangleleft}\overline{\Sigma} MA \xrightarrow{{\triangleleft}\mathsf{cons}} {\triangleleft} MA\right)$$

$$\lambda^{\triangleleft} = \left(\overline{\Gamma}{\triangleleft\triangleleft} MA = {\triangleleft}\overline{\Gamma}{\triangleleft} MA \xrightarrow{{\triangleleft}\mathsf{cons}} {\triangleleft} MA\right)$$

$$\lambda^{\triangleright} = \left({\triangleright\triangleleft} MA \xrightarrow{\epsilon} MA = {\triangleleft\triangleright} MA \xrightarrow{{\triangleleft}\mathsf{cons}} {\triangleleft} MA\right)$$

The bracketing operation is then defined as $M = M{\triangleleft\triangleright} \xrightarrow{\lambda} {\triangleleft} M {\triangleright}$.

The desired natural isomorphism $i : E \to \downharpoonleft M \upharpoonright$ is the unique morphism from the initial algebra induced by the natural transformation $[i^{\mathrm{var}}, i^{\Sigma}, i^{\Gamma}] : \mathsf{Id} + \Sigma \downharpoonleft M \upharpoonright + \Gamma \downharpoonleft M \upharpoonright \downharpoonleft M \upharpoonright \to \downharpoonleft M \upharpoonright$, where:

$$i^{\mathrm{var}} = \left(\mathsf{Id} \xrightarrow{\eta} \downharpoonleft\upharpoonright \xrightarrow{\downharpoonleft \eta} \downharpoonleft M \upharpoonright\right) \qquad i^{\Sigma} = \left(\Sigma \downharpoonleft M \upharpoonright = \downharpoonleft \overline{\Sigma} M \upharpoonright \xrightarrow{\downharpoonleft\,\mathsf{cons}} \downharpoonleft M \upharpoonright\right)$$

$$i^{\Gamma} = \Big(\Gamma \downharpoonleft M \upharpoonright \downharpoonleft M \upharpoonright \xrightarrow{\Gamma \downharpoonleft M \epsilon} \Gamma \downharpoonleft MM \upharpoonright = \Gamma \downharpoonleft M {\triangleleft\triangleright} M \upharpoonright \xrightarrow{\Gamma \downharpoonleft \lambda} \Gamma \downharpoonleft {\triangleleft} M {\triangleright} M \upharpoonright =$$

$$\downharpoonleft \overline{\Gamma}{\triangleleft} M {\triangleright} M \upharpoonright \xrightarrow{\downharpoonleft\,\mathsf{cons}\,\mathsf{cons}} \downharpoonleft MM \upharpoonright \xrightarrow{\downharpoonleft \mu} \downharpoonleft M \upharpoonright\Big)$$

As the next step, we define the inverse of i. As an auxiliary structure, for an endofunctor G on the category $\mathscr{C}$, we define a functor $G^{+} : \mathscr{C} \to \mathscr{C}^{|\mathbb{N}|}$ given on objects as:

$$(G^{+}A)_n = G^{n+1}A$$

The action on morphisms is obvious. (Note that the G^{+} construction is relevant not only to this proof, since it appears, for appropriate endofunctors G, as carriers in Examples 4.2 and 4.5.) Using the fact that $MA = \mu X. A + \overline{\Sigma} X + \overline{\Gamma}{\triangleleft} X + {\triangleright} X$, we define the following natural transformation $k : M \upharpoonright A \to E^{+}A$ as the unique morphism from the initial algebra induced by the morphism $[k^{\mathrm{var}}, k^{\Sigma}, k^{\triangleleft}, k^{\triangleright}]$, where:

$$k^{\mathrm{var}}_0 = \left((\upharpoonright A)_0 = A \xrightarrow{\eta} EA = (E^{+}A)_0\right)$$

$$k^{\mathrm{var}}_{n+1} = \left((\upharpoonright A)_{n+1} = 0 \xrightarrow{!} E^{n+2}A = (E^{+}A)_{n+1}\right)$$

$$k^{\Sigma}_n = \left((\overline{\Sigma} E^{+}A)_n = \Sigma E E^n A \xrightarrow{\mathsf{cons}} E E^n A = (E^{+}A)_n\right)$$

$$k^{\triangleleft}_n = \Big((\overline{\Gamma}{\triangleleft} E^{+}A)_n = (\overline{\Gamma} E^{+}A)_{n+1} = \Gamma E E E^n A$$

$$\xrightarrow{\mathsf{cons}} E E^n A = (E^{+}A)_n\Big)$$

$$k^{\triangleright}_0 = \left(({\triangleright} E^{+}A)_0 = 0 \xrightarrow{!} (E^{+}A)_0\right)$$

$$k^{\triangleright}_{n+1} = \Big(({\triangleright} E^{+}A)_{n+1} = (E^{+}A)_n = E^{n+1}A$$

$$\xrightarrow{\eta} E E^{n+1}A = E^{n+2}A = (E^{+}A)_{n+1}\Big)$$

The inverse of i is obtained by projecting k on $\mathscr{C}$:

$$i^{-1}A = \left(\downharpoonleft M \upharpoonright A \xrightarrow{\downharpoonleft kA} \downharpoonleft E^{+}A = (E^{+}A)_0 = EA\right)$$

To see that $\downharpoonleft k \cdot i = \mathsf{id}$, it is enough to show that $\downharpoonleft k$ is a morphism between algebras, that is, $\downharpoonleft k \cdot [i^{\mathrm{var}}, i^{\Sigma}, i^{\Gamma}] = \mathsf{in} \cdot (\mathsf{id} + \Sigma \downharpoonleft k + \Gamma \downharpoonleft k \downharpoonleft k)$. (This is the so-called *fusion law*; see, for example, Fokkinga's PhD thesis [11].) In the other direction, $i \cdot \downharpoonleft k = \mathsf{id}$, the situation is a bit more subtle, as $\downharpoonleft k$ is not a morphism from an initial algebra. As a workaround, we can generalise i to a morphism $q : E^{+}A \to M \upharpoonright A$ such that $q \cdot k = \mathsf{id}$ (which can be shown using the fusion law) and $\downharpoonleft q = i$. Putting this together we obtain $i \cdot \downharpoonleft k = \downharpoonleft q \cdot \downharpoonleft k = \downharpoonleft (q \cdot k) = \downharpoonleft \mathsf{id} = \mathsf{id}$.

Remark 5.2. As expected, the adjunction $F{\upharpoonleft} \dashv {\downharpoonleft}U$ is in general not monadic. Moreover, the comparison functor is in general neither full nor faithful.

6 Haskell Implementation

In this section we show how the abstract categorical ideas can guide a Haskell implementation of effects with scoping operations. Of the two monads we have presented, E is the most amenable to implementation. For instance, we can represent it as follows in Haskell. The implementation of E is called *Prog*, and the type variables f and g are placeholders for the endofunctors Σ and Γ respectively.

```
data Prog f g a = Var a
                | Op (f (Prog f g a))
                | Scope (g (Prog f g (Prog f g a)))

instance (Functor f, Functor g) ⇒ Monad (Prog f g) where
  return = Var
  Var x    ≫= f = f x
  Op op    ≫= f = Op (fmap (≫=f) op)
  Scope sc ≫= f = Scope (fmap (fmap (≫=f)) sc)
```

The implementation of the monad instance is similar to the usual free monad implementation, except for the additional case of *Scope*, where the recursive call is under two layers.

For example, we define *NDProg*, the monad for programs that use nondeterminism and **once**:

```
data Choice a = Fail | Or a a deriving Functor
data Once a   = Once a deriving Functor
type NDProg  = Prog Choice Once
```

Now we can express the program from Example 4.4 in Haskell as

```
example₄ :: NDProg Int
example₄ = do x ← once (or (return 1) (return 5))
              or (return x) (return (x + 1))
```

using these three smart constructor functions:

```
fail :: NDProg a
fail = Op Fail

or :: NDProg a → NDProg a → NDProg a
or x y = Op (Or x y)

once :: NDProg a → NDProg a
once x = Scope (Once (fmap return x))
```

As Section 4.1 observes, the right notion of interpretation for *Prog f g* is not apparent. Fortunately, by way of the ${\downharpoonleft}M{\upharpoonleft}$ monad we can derive the appropriate type of algebras, *Alg*:

```
data Nat = Zero | Succ Nat

data Alg f g a = A { a :: ∀n. f (a n)        → a n
                   , d :: ∀n. g (a (Succ n)) → a n
                   , p :: ∀n. a n            → a (Succ n) }
```

The three functions involve a *Nat*-indexed carrier type. This use of indexed types is a complicating factor, which requires the non-standard GHC-Haskell `DataKinds` extension. It also shows up in the *fold* function that uses an algebra to interpret a whole program.

```
fold :: (Functor f, Functor g) ⇒ Alg f g a → Prog f g (a n) → a n
fold alg (Var x)    = x
fold alg (Op op)    = a alg (fmap (fold alg) op)
fold alg (Scope sc) =
  d alg (fmap (fold alg ∘ fmap (p alg ∘ fold alg)) sc)
```

For practical use, it is convenient to use a *gen*erator function to turn a program's unindexed return type r into the indexed carrier type *a Zero* before folding over the structure.

```
run :: (Functor f, Functor g)
    ⇒ (r → a Zero) → Alg f g a → (Prog f g r → a Zero)
run gen alg prog = fold alg (fmap gen prog)
```

For our nondeterminism example we require a carrier type $Carrier_{ND}$:: $* \to Nat \to *$ such that $Carrier_{ND}\ a\ Zero \cong [a]$, $Carrier_{ND}\ a\ (Succ\ Zero) \cong [[a]]$, and so on. That is, the carrier should be such that $Carrier_{ND}\ a\ n \cong [a]^{n+1}$. This datatype can be represented in Haskell as:

```
data Carrier_ND a n = ND [Carrier_ND' a n]

data Carrier_ND' a :: Nat → ∗ where
  CZ_ND :: a → Carrier_ND' a Zero
  CS_ND :: [Carrier_ND' a n] → Carrier_ND' a (Succ n)
```

The corresponding generator and algebra that give semantics to the syntax defined in $example_4$ using $Carrier_{ND}$ are:

```
gen_ND :: a → Carrier_ND a Zero
gen_ND x = ND [CZ_ND x]

alg_ND :: Alg Choice Once (Carrier_ND a)
alg_ND = A {..} where
  a :: ∀n a. Choice (Carrier_ND a n) → Carrier_ND a n
  a Fail                 = ND []
  a (Or (ND l) (ND r))   = ND (l ++ r)

  d :: ∀n a. Once (Carrier_ND a (Succ n)) → Carrier_ND a n
  d (Once (ND []))       = ND []
  d (Once (ND (CS_ND l : _))) = ND l

  p :: ∀n a. Carrier_ND a n → Carrier_ND a (Succ n)
  p (ND l)               = ND [CS_ND l]
```

Now *main* ties everything together to interpret $example_4$.

```
main :: [Int]
main = toList (run gen_ND alg_ND example₄) where
  toList :: Carrier_ND a Zero → [a]
  toList (ND l) = map (λ(CZ_ND x) → x) l
```

which does indeed yield the expected result $[1, 2]$.

7 More Examples

In this section, to demonstrate that our framework is quite expressive, we show how to use it to model two computational effects that rely heavily on scoped operations: state with local variables and concurrency. We express these examples as Haskell implementations, since we believe that they are more readable this way, and it shows that our implementation is indeed applicable in practice.

7.1 State with Local Variables

The first example is given by state with local variables. It has two algebraic operations: *Put x s k* that assigns the value s to the variable x, and continues with the computation k, and *Get x f* that retrieves the value stored in the variable x and applies the function f to the result in order to obtain the value to proceed

with. There is one scoped operation *Local x s k* that creates a fresh variable x, which initially stores the value s. If there already is a variable called x, it is shadowed in k by the fresh x, but outside of the scope created by *Local*, the old x is used.

We define the signature together with some smart constructors:

```
type Name      = String
data State s a = Get Name (s → a) | Put Name s a deriving Functor
data Local s a = Local Name s a deriving Functor
type LSProg s = Prog (State s) (Local s)

get :: Name → LSProg s s
get x = Op (Get x (λs → return s))

put :: Name → s → LSProg s ()
put x s = Op (Put x s (return ()))

local :: Name → s → LSProg s a → LSProg s a
local x s p = Scope (fmap (fmap return) (Local x s p))
```

The state that we use in our example is modelled by a function from variable names to values. We also define two auxiliary functions:

```
type Memory s = Name → Maybe s

retrieve :: Name → Memory s → s
retrieve x m = case m x of Just s   → s
                           Nothing → error ("var undefined")

update :: Name → s → Memory s → Memory s
update x s m y | x ≡ y      = Just s
               | otherwise  = m y
```

The carrier that we use is defined as a composition of n state transformers $Memory\ s \rightarrow (a, Memory\ s)$. To avoid complicating the carrier *retrieve* does not wrap s in a *Maybe* to indicate the *error*. The promotion morphism wraps the carrier in a transformer given by the identity on states, while the demotion morphism composes the two outer transformers, being careful to use the original value for the local variable.

```
data Carrier_LS s a n =
  LS {run_LS :: (Memory s → (Carrier'_LS s a n, Memory s))}
data Carrier'_LS s a :: Nat → ∗ where
  CZ_LS :: a → Carrier'_LS s a Zero
  CS_LS :: (Memory s → (Carrier'_LS s a n, Memory s)) →
           Carrier'_LS s a (Succ n)

gen_LS :: a → Carrier_LS s a Zero
gen_LS a = LS (λm → (CZ_LS a, m))

alg_LS :: Alg (State s) (Local s) (Carrier_LS s a)
alg_LS = A {..} where
  a (Put x s (LS f)) = LS (f ∘ update x s)
  a (Get x p)        = LS (λm → run_LS (p (retrieve x m)) m)
  d :: ∀s n a. Local s (Carrier_LS s a (Succ n)) → Carrier_LS s a n
  d (Local x s (LS f)) = LS (λm → case f (update x s m) of
    (CS_LS g, n) → g (update x (retrieve x m) n))
  p :: ∀s n a. Carrier_LS s a n → Carrier_LS s a (Succ n)
  p l = LS (λm → (CS_LS (run_LS l), m))
```

We define a function that allows us to test computations. It runs a program starting with empty memory:

```
test_LS :: LSProg s a → a
test_LS p = case fst (run_LS (run gen_LS alg_LS p) (λx → Nothing)) of
              CZ_LS a → a

incr :: Name → Int → LSProg Int ()
incr x i = do
  v ← get x
  put x (v + i)

testProg_LS :: LSProg Int (Int, Int)
testProg_LS = do put "x" 1
                 put "y" 1
                 local "x" 100 (do incr "x" 100
                                   v ← get "x"
                                   incr "y" v)
                 incr "x" 2
                 incr "y" 2
                 vx ← get "x"
                 vy ← get "y"
                 return (vx, vy)
```

We can test the program above by calling $test_{LS}\ testProg_{LS}$ and obtain $(3, 203)$.

7.2 Concurrency via Resumptions

We show how one can perform concurrent computations in any monad m; compare [7, 21]. The algebraic signature consists of two operations: *Act c*, which performs any action $c :: m\ a$, and *Kill*, which ends the current process without returning a value. The scoped signature consists of *Spawn c c′*, which executes the programs c and c' in parallel, and returns the result value of c, and *Atomic c*, which guarantees that the computation c is not interleaved with any other computation. We encode this signature together with some smart constructors:

```
data Act m a = Act (m a) | Kill deriving Functor
data Con a = Spawn a a | Atomic a deriving Functor
type ConProg m = Prog (Act m) Con

lift :: Functor m ⇒ m a → ConProg m a
lift m = Op (Act (fmap return m))

kill :: ConProg m a
kill = Op Kill

spawn :: Functor m ⇒ ConProg m a → ConProg m b → ConProg m a
spawn p q = Scope (fmap (fmap return) (Spawn p (q ≫ kill)))
atomic :: Functor m ⇒ ConProg m a → ConProg m a
atomic p = Scope (fmap (fmap return) (Atomic p))
```

The semantics is based on the *Resumption* datatype, which is simply the free monad generated by m as a functor. It comes with two auxiliary functions: *retraction* flattens the computation by multiplying out all the layers of a *Resumption* value, while *interleaveL* allows us to compose two resumptions in parallel by interleaving the layers:

```
data Resumption m a = More (m (Resumption m a)) | Done a
  deriving Functor

retraction :: Monad m ⇒ Resumption m a → m a
retraction (More m)   = m ≫= retraction
retraction (Done a)   = return a
interleaveL :: Functor m ⇒ Resumption m a → Resumption m b →
```

```
                  Resumption m a
interleaveL (Done a)  r             = fmap (λ_ → a) r
interleaveL r         (Done _)      = r
interleaveL (More m) (More m') =
  More (fmap (λr → More (fmap (λr' → interleaveL r r') m')) m)
```

The carrier of the semantics is given by *Resumption* composed with itself an appropriate number of times. The algebra collects the *m*-actions in a resumption. The function *interleaveL* is used to interpret *Spawn* operations, while *retraction* guarantees that atomic operations are performed without any interleaving.

```
data CarrierCon m a n =
  CC {run_CC :: Resumption m (CarrierCon' m a n)}
data CarrierCon' m a :: Nat → ∗ where
  CZ_CC :: a → CarrierCon' m a Zero
  CS_CC :: Resumption m (CarrierCon' m a n) →
             CarrierCon' m a (Succ n)
rJoin :: Functor m ⇒ Resumption m (CarrierCon' m a (Succ n))
                    → Resumption m (CarrierCon' m a n)
rJoin (Done (CS_CC r)) = r
rJoin (More m)         = More (fmap rJoin m)
gen_CC :: Monad m ⇒ a → CarrierCon m a Zero
gen_CC a = CC (Done (CZ_CC a))
alg_CC :: Monad m ⇒ Alg (Act m) Con (CarrierCon m a)
alg_CC = A {..} where
  a (Act m) = CC (More (fmap run_CC m))
  a (Kill)  = CC (Done (error "main process killed"))
  d :: ∀m n a. Monad m ⇒ Con (CarrierCon m a (Succ n)) →
         CarrierCon m a n
  d (Atomic (CC r)) =
    CC (More (fmap (λ(CS_CC s) → s) (retraction r)))
  d (Spawn (CC r) (CC r')) = CC (rJoin (interleaveL r r'))
  p :: ∀m n a. CarrierCon m a n → CarrierCon m a (Succ n)
  p (CC r) = CC (Done (CS_CC r))
```

We can test the syntax and semantics. We use the *IO* monad for *m*:

```
runCon :: Monad m ⇒ ConProg m a → m a
runCon p = retraction
  (fmap (λ(CZ_CC a) → a) (run_CC (run gen_CC alg_CC p)))
say :: String → ConProg IO ()
say = lift ∘ putStr
conTest_1 :: ConProg IO ()
conTest_1 = do
  spawn (say "hello " ≫ say "world ")
        (say "goodbye " ≫ say "cruel " ≫ say "world ")
```

The computation *runCon conTest*$_1$ prints out `hello goodbye world cruel world`. We can also nest scoped operations:

```
conTest_2 :: ConProg IO ()
conTest_2 = do
  spawn (atomic (spawn (mapM say ["a", "b", "c"])
                       (mapM say ["A", "B", "C"])))
        (atomic (spawn (mapM say ["1", "2", "3"])
                       (mapM say ["-", "-", "-"])))
```

The computation *runCon conTest*$_2$ prints out `aAbBcC1-2-3-`.

8 Discussion

More discussion on practical motivation for scoped operations can be found in Wu *et al.* [29]. One can also compare with explicit scopes for binding constructs, as in Hendriks and van Oostrom's [15] ⅄-calculus or Gabbay *et al.*'s [12] explicit name management in the nominal setting.

Our monad E is a generalisation of Ghani *et al.*'s monad of explicit substitutions [13] defined using higher-order syntax, that is, the initial algebra semantics in $\mathscr{C}^{\mathscr{C}}$. Higher-order syntax is a powerful tool, useful also to formalise variable binding, or to model nested datatypes in functional programming [16]. Our generalisation is very slight, as the only addition is the endofunctor Γ, but our subsequent results shed some new light on Ghani *et al.*'s original monad $E^{\Sigma,\mathsf{Id}}$. For example, Ghani *et al.* define a 'flattening' morphism, which evaluates explicit substitutions to actual substitutions, called $\mathsf{resolve}A : E^{\Sigma,\mathsf{Id}}A \to \Sigma^* A$. Thanks to the second presentation of E as $\lfloor M \rceil$, which arises as the monad induced by scoped algebras, this morphism can be obtained in a more principled fashion, as arising from the scoped algebra $\langle \overline{\Sigma^* A}, \mathsf{cons}, \mathsf{id}, \mathsf{id} \rangle$, where $(\overline{\Sigma^* A})_i = \Sigma^* A$.

Moreover, Ghani *et al.* [13] comment on the apparent lack of modularity in the higher-order syntax for explicit substitutions, at least in terms of coproducts in the category of monads. In particular, the explicit substitution monad $E^{\Sigma,\mathsf{Id}}$ is in general not a coproduct of Σ^* and $E^{K_0,\mathsf{Id}}$. However, thanks to our presentation of E as $\lfloor M \rceil$, we can see some modularity, but only modulo sandwiching in the $\lceil \dashv \lfloor$ adjunction. In detail, notice that $\Sigma^* = \lfloor (\overline{\Sigma})^* \rceil$ and $E^{K_0,\mathsf{Id}} = \lfloor (\triangleleft + \triangleright)^* \rceil$, while $E^{\Sigma,\mathsf{Id}} = \lfloor (\overline{\Sigma} + \triangleleft + \triangleright)^* \rceil$, and $(\overline{\Sigma} + \triangleleft + \triangleright)^*$ is obviously the coproduct of $(\overline{\Sigma})^*$ and $(\triangleleft + \triangleright)^*$.

Note, however, that if we wanted two separate sets of brackets, or, more generally, two sets of scoped operations, each with its own kind of closing brackets, it would not be enough to define $\blacktriangleleft = \triangleleft$ and $\blacktriangleright = \triangleright$, and the monad $\lfloor \overline{\Sigma} + \overline{\Gamma}\triangleleft + \triangleright + \overline{\Gamma'}\blacktriangleleft + \blacktriangleright \rceil$, as it would be possible to close $\overline{\Gamma}\triangleleft$ with $\blacktriangleright$, so the brackets would not match. To consider such structures, we can augment the level information from $\mathbb{N}$ to a list of some data, for example, to capture what kind of scope there is at each level. This could be useful for modularity, when we have n different effects in play. In such a case, we could use objects indexed by lists of labels coming from the set $\{1, \ldots, n\}$, denoting the effect to which a particular scope belongs. One could also try to replace $\triangleright \dashv \triangleleft$ with other adjunctions. We leave filling in the details as future work.

One can think of other variations on the problem and the solution discussed in this paper. For example, we can generalise the signatures Σ and Γ to $\mathbb{N}$-indexed functors, which expresses that different operations are available at different levels. For instance, we can allow throwing exceptions only inside a **catch**, or we can allow throwing to a particular **catch** above, not necessarily the immediately surrounding one. This is related to the more general form of indexed monads [19].

An alternative monad that one could use to formalise syntax with scoped operations is to allow variables in scopes. This can be obtained with the monad given by the following endofunctor, defined as a carrier of an initial algebra over $\mathscr{C}^{\mathscr{C}}$:

$$T = \mu H.\, \mathsf{Id} + \Sigma H + \Gamma H(\mathsf{Id} + H)$$

Eilenberg-Moore algebras of T need to respect structural properties in the scope, but it is not clear whether they have a nicer, 'algebraic' presentation along the lines of the monad M. Moreover, variables in scopes do not seem to add anything substantial to the examples that we have considered.

As future work, one can extend the presented material in both theoretical and practical aspects. As for the former, it would be interesting to fill in some details that we gloss over in this paper, such as conditions for existence of the initial algebras (for example, Ghani *et al.* focus on finitary endofunctors on locally finitely presentable categories) or strength (that is why we restrict the particular examples to **Set**, in which every endofunctor is canonically strong). Another issue is to establish a relationship between syntax with scopes and frameworks like the second-order algebraic theories introduced by Fiore and Mahmoud [9]. As for the practical aspects, it is an interesting task to fully work out an implementation with more advanced features, such as fusion [28], either in Haskell or a language with native support for handlers, like Eff [4].

Moreover, we notice that our definition of operation does not capture all of the different kinds of syntax used in practice. In particular, one sometimes wants an operation to be extranatural. Examples include Benton and Kennedy's [5] 'exceptional syntax', or concurrency in which different threads return values of different types (compare Wu *et al.* [29]). From the syntactic side, such operations are easily definable using the initial-algebra semantics in $\mathscr{C}^{\mathscr{C}}$, while working out a systematic 'handler' part is a matter of future work.

Acknowledgments

The authors would like to thank Marcelo Fiore for enlightening discussions, and the anonymous reviewers for their constructive comments regarding presentation.

This work is partially supported by the National Science Centre, Poland under POLONEZ 3 grant no. 2016/23/P/ST6/02217, by the Research Foundation - Flanders, and by Agencia Nacional de Promoción Científica y Tecnológica (PICT-2016-0464).

References

[1] Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. 2014. Relative Monads Formalised. *Journal of Formalized Reasoning* 7, 1 (2014), 1–43. https://doi.org/10.6092/issn.1972-5787/4389
[2] Roland Carl Backhouse, Marcel Bijsterveld, Rik van Geldrop, and Jaap van der Woude. 1995. Categorical Fixed Point Calculus. In *Category Theory and Computer Science (Lecture Notes in Computer Science (LNCS))*, David H. Pitt, David E. Rydeheard, and Peter Johnstone (Eds.), Vol. 953. Springer, 159–179.
[3] Michael Barr. 1970. Coequalizers and free triples. *Mathematische Zeitschrift* 116, 4 (1970), 307–322. https://doi.org/10.1007/BF01111838
[4] Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *The Journal of Logic and Algebraic Programming* 84, 1 (2015), 108–123. https://doi.org/10.1016/j.jlamp.2014.02.001
[5] Nick Benton and Andrew Kennedy. 2001. Exceptional Syntax. *Journal of Functional Programming* 11, 4 (2001), 395–410. https://doi.org/10.1017/S0956796801004099
[6] Edwin Brady. 2013. Programming and reasoning with algebraic effects and dependent types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional programming*. ACM, 133–144.
[7] Koen Claessen. 1999. A Poor Man's Concurrency Monad. *Journal of Functional Programming* 9, 3 (May 1999), 313–323. https://doi.org/10.1017/S0956796899003342
[8] Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. 2015. Effective Concurrency with Algebraic Effects. In *OCaml Users and Developers Workshop*.
[9] Marcelo P. Fiore and Ola Mahmoud. 2010. Second-Order Algebraic Theories - (Extended Abstract). In *Mathematical Foundations of Computer Science 2010, 35th International Symposium, MFCS 2010, Brno, Czech Republic, August 23-27, 2010. Proceedings (Lecture Notes in Computer Science)*, Petr Hlinený and Antonín Kucera (Eds.), Vol. 6281. Springer, 368–380. https://doi.org/10.1007/978-3-642-15155-2_33
[10] Marcelo P. Fiore, Gordon D. Plotkin, and Daniele Turi. 1999. Abstract Syntax and Variable Binding. In *14th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, 193–202.
[11] Martinus Maria Fokkinga. 1992. *Law and Order in Algorithmics*. Ph.D. Dissertation. University of Twente, Enschede. http://doc.utwente.nl/66251/
[12] Murdoch J. Gabbay, Dan R. Ghica, and Daniela Petrisan. 2015. Leaving the Nest: Nominal Techniques for Variables with Interleaving Scopes. In *24th EACSL Annual Conference on Computer Science Logic (CSL 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Stephan Kreutzer (Ed.), Vol. 41. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 374–389. https://doi.org/10.4230/LIPIcs.CSL.2015.374
[13] Neil Ghani, Tarmo Uustalu, and Makoto Hamana. 2006. Explicit substitutions and higher-order syntax. *Higher-Order and Symbolic Computation* 19, 2-3 (2006), 263–282. https://doi.org/10.1007/s10990-006-8748-4
[14] Jeremy Gibbons and Ralf Hinze. 2011. Just do it: simple monadic equational reasoning. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). ACM, 2–14. https://doi.org/10.1145/2034773.2034777
[15] Dimitri Hendriks and Vincent van Oostrom. 2003. adbmal. In *Automated Deduction - CADE-19, 19th International Conference on Automated Deduction Miami Beach, FL, USA, July 28 - August 2, 2003, Proceedings (Lecture Notes in Computer Science)*, Franz Baader (Ed.), Vol. 2741. Springer, 136–150. https://doi.org/10.1007/978-3-540-45085-6_11
[16] Patricia Johann and Neil Ghani. 2007. Initial Algebra Semantics Is Enough!. In *Typed Lambda Calculi and Applications, 8th International Conference, TLCA (Lecture Notes in Computer Science)*, Simona Ronchi Della Rocca (Ed.), Vol. 4583. Springer, 207–222. https://doi.org/10.1007/978-3-540-73228-0_16
[17] Oleg Kiselyov, Amr Sabry, and Cameron Swords. 2013. Extensible Effects: An Alternative to Monad Transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell (Haskell '13)*. ACM, 59–70.
[18] Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do be do be do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 500–514. https://doi.org/10.1145/3009837
[19] Conor McBride. 2011. Kleisli arrows of outrageous fortune. (2011). https://personal.cis.strath.ac.uk/conor.mcbride/Kleisli.pdf.
[20] Eugenio Moggi. 1989. *An abstract view of programming languages*. Technical Report ECS-LFCS-90-113. Edinburgh University, Department of Computer Science.
[21] Maciej Piróg and Jeremy Gibbons. 2012. Tracing monadic computations and representing effects. In *Proceedings Fourth Workshop on Mathematically Structured Functional Programming, Tallinn, Estonia, 25 March 2012 (Electronic Proceedings in Theoretical Computer Science)*, Vol. 76. Open Publishing Association, 90–111. https://doi.org/10.4204/EPTCS.76.8
[22] Gordon D. Plotkin and John Power. 2002. Notions of Computation Determine Monads. In *Foundations of Software Science and Computation Structures (LNCS)*, Vol. 2303. Springer, 342–356.
[23] Gordon D. Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11, 1 (2003), 69–94. https://doi.org/10.1023/A:1023064908962
[24] Gordon D. Plotkin and John Power. 2004. Computational Effects and Operations: An Overview. *Electronic Notes in Theoretical Computer Science* 73 (2004), 149–163. https://doi.org/10.1016/j.entcs.2004.08.008
[25] Gordon D. Plotkin and Matija Pretnar. 2009. *Programming Languages and Systems: 18th European Symposium on Programming, ESOP 2009, Proceedings*. Springer, Chapter Handlers of Algebraic Effects, 80–94.
[26] Tom Schrijvers, Nicolas Wu, Benoit Desouter, and Bart Demoen. 2014. Heuristics Entwined with Handlers Combined: From Functional Specification to Logic Programming Implementation. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming, Kent, Canterbury, United Kingdom, September 8-10, 2014*, Olaf Chitil, Andy King, and Olivier Danvy (Eds.). ACM, 259–270. https://doi.org/10.1145/2643135.2643145
[27] Wouter Swierstra. 2008. Data types à la carte. *Journal of Functional Programming* 18, 4 (2008), 423–436. https://doi.org/10.1017/S0956796808006758
[28] Nicolas Wu and Tom Schrijvers. 2015. Fusion for Free - Efficient Algebraic Effect Handlers. In *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings (Lecture Notes in Computer Science)*, Ralf Hinze and Janis Voigtländer (Eds.), Vol. 9129. Springer, 302–322. https://doi.org/10.1007/978-3-319-19797-5_15
[29] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect Handlers in Scope. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell (Haskell '14)*. ACM, New York, NY, USA, 1–12.