Algebras for Effectful Program-Environment Interactions [*]

Niels F.W. Voorneveld

Tallinn University of Technology, Tallinn, Estonia
niels.voorneveld@taltech.ee

Programs never run in isolation. They are evaluated within an environment, which consists of program modules, compilers, operating systems, hardware and human users. Programs may invoke queries to such environments; operations which are resolved externally and can be considered effectful. We will study a categorical way of interpreting the behaviour of such effectful operations using algebras.

In the first part, we define how to describe denotations of programs and environments interacting with each other using queries. Such interactions can then be modelled using *monad-comonad interaction laws* as featured in [1]. In the second part, we will use algebras and coalgebras to specify behaviour of programs and environments, and use the results from [2] to describe whole systems of interacting components with a single algebra.

Denotational models for effect operations

Programs are seen as active components of computation, evaluating to a result. While being evaluated, a program may invoke effectful operations given by a *signature*, also known as a *container*. Such a signature of operations is given by a set S with elements $\mathsf{op} \in S$ each with an associated arity $ar(\mathsf{op})$ given by a set. When $\mathsf{op} \in S$ is invoked, it sends a request to the environment: it asks the environment to choose and provide an element from the set $ar(\mathsf{op})$. Depending on the response, the program may continue in a variety of ways.

Such programs can be denoted by elements of a monad in the category of sets **Set**, in particular the free monad T_S over the endofunctor $F_S X = \Sigma_{\mathsf{op} \in S} X^{ar(op)}$. This monad is inductively defined, where its elements are either given by results $\langle x \rangle \in T_S X$ where $x \in X$, or requests with continuations $\mathsf{op}(c) \in T_S X$ where $\mathsf{op} \in S$ and $c : ar(\mathsf{op}) \to T_S X$. We are not yet making any assumptions on how these operations should be resolved. We will see later on how requests can be answered by both direct interaction with a stateful environment, or by a quantitative assessment of possibilities using algebras.

Dually, we model an environment capable of answering queries from the signature S. We see such an environment as passive, carrying an internal state which may change when something interacts with the environment. We model this using a cofree comonad D_S over the functor $G_S Y = \Pi_{\mathsf{op} \in S}\, ar(\mathsf{op}) \times Y$. This comonad is coinductively defined, where each element b of $D_S Y$ contains an internal state $\varepsilon^S_Y(b) \in Y$ and for each operation $\mathsf{op} \in S$ a response and continuation $\mathsf{op}(b) \in ar(\mathsf{op}) \times D_S Y$. Though in general, $D_S Y$ contains any possible way of resolving queries, we will later limit the possible behaviour patterns, either by communication with an even higher environment, or by formulating concrete specifications of behaviour patterns using coalgebras.

Queries from programs modelled by $T_S X$ can be directly resolved by answers from environments modelled by $D_S Y$. For example, $\mathsf{op}(c) \in T_S X$ raises query op, and $b \in D_S Y$ can answer with $\mathsf{op}(b) = (i, b') \in ar(\mathsf{op}) \times D_S Y$, and the program will continue by evaluating $c(i) \in T_S X$ in environment $b' \in D_S Y$. We do this until the program provides a result $\langle x \rangle$ with $x \in X$, at

[*]This work is based on collaborations and discussions with Tarmo Uustalu, partially featured in [2].
The author was supported by the ESF funded Estonian IT Academy research measure (project 2014-2020.4.05.19-0001) and the Estonian Research Council grant PSG659.

which point we can read the final state $\varepsilon(b) \in Y$ from the environment. This gives a transformation $\mathsf{d}^S_{X,Y} : T_S X \times D_S Y \to X \times Y$, natural in both X and Y, known as a *monad-comonad interaction law* [1]. This models an instance of direct program-environment interaction.

We would like to consider more general situations, for example when not all queries are resolved, or when the interaction itself may invoke new queries to another environment higher up. We take three signatures S, P, R, where a program may invoke queries from S, an environment can answer queries from P, and there may be residual queries from R to a higher-level environment. The following definition is an instantiation of a more general definition in [1].

Definition 1. Given signatures S, P, R, a *T_R-residual monad-comonad interaction law* between T_S and D_P is a natural transformation $\phi : T_S X \times D_P Y \to T_R(X \times Y)$ satisfying a unit and a multiplication equation.

We can split a signature into two parts, resolving one part with an environment, and leaving the other part untouched. We use the disjoint union $+$ signatures to describe such a combination of two signatures, and can formulate an interaction law $\mathsf{d}^{S,R}_{X,Y} : T_{S+R} X \times D_S Y \to T_R(X \times Y)$ which only resolves queries from S. A T_R-residual monad-comonad interaction law ϕ between T_S and D_P can be build by composing a *monad morphism* $m^\phi : T_S \to T_{P+R}$ with the interaction law $\mathsf{d}^{P,R}_{X,Y} : T_{P+R} X \times D_P Y \to T_R(X \times Y)$. Hence we can code up interaction laws using families of programs denoted by $\Pi_{\mathsf{op} \in S} T_{P+R}(ar(\mathsf{op}))$. Not all interaction laws can be build this way, since some interaction laws may backtrack to an earlier environment state.

Using interaction laws, it is possible to create whole networks of interacting programs and environments. We would like to talk about the behaviour of programs within such networks, using algebras to formulate behavioural properties of those programs.

Algebraic descriptions of effect behaviour

The practicality of algebras is two-fold. Firstly, they can be used to abstractly describe networks of environments, and use them to formulate stateful predicates on programs. Secondly, they may be used to describe the behaviour of queries which cannot be directly resolved by a computation in the environment, for instance probabilistic queries.

Algebras can be used to formulate quantitative predicates for effectful programs [3]. This generalises yes-or-no questions such as: "Does this program produce an even number?", to more open-ended question such as: "How long does it take before..." or "What is the probability that...". Concretely, we use a *monad algebra* over T_S; a function $\alpha : T_S A \to A$ over some set of answers A, satisfying a unit and multiplication equation. Monad algebras α over T_S are in bijection with families of *local functions* $\widehat{\alpha} \in \Pi_{\mathsf{op} \in S}(A^{ar(\mathsf{op})} \to A)$. Moreover, we can lift predicates on results $P : X \to A$ to predicates on programs $\alpha \circ T_S(P) : T_S X \to A$. Examples:

Time: We can model *execution time* by using a signature $S = \{\mathsf{tick}\}$ with one query of arity $\{*\}$ (so only one possible response to the query). $T_S X$ is isomorphic to $\mathbb{N} \times X$, and we can define an algebra $\alpha : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ using addition (coincidentally a free algebra).

Probability: We can model *probabilistic programs* using a single binary operation $S = \{\mathsf{por}\}$ of arity $\{0, 1\}$. The set $T_S X$ contains binary trees, which we give the algebra $P : T_S[0,1] \to [0,1]$ over the real number interval $[0,1]$, locally defined using $(a, b) \mapsto (a + b)/2$.

Environments may be specified using comonad coalgebras over D_P; a function $\beta : B \to D_P B$ over some set of generators B satisfying a co-unit and co-multiplication equation. These generate patterns of behaviour; possible ways environments can react to queries. Comonad coalgebras β over D_P are in bijection with families of local functions $\widehat{\beta} \in \Pi_{\mathsf{op} \in P}(B \to ar(\mathsf{op}) \times B)$ specifying both the responses to and resulting state after a query.

Input: We model any possible way of responding to a query $P : \{\mathsf{read}\}$ of arity $ar(\mathsf{read}) = D$. This can be done using infinite streams $D^{\mathbb{N}}$, with a coalgebra $\beta : D^{\mathbb{N}} \to D_P D^{\mathbb{N}}$ given by the local function $f \mapsto (f(0), \lambda n.f(n+1))$ (which is a cofree coalgebra).

Global store: Now let us read and write from a persistent global store, with a signature $P = \{\mathsf{read}\} \cup \{\mathsf{write}_n : n \in D\}$ which adds to read a set queries write_n of arity $\{*\}$. We define the coalgebra over D, with local functions: $\mathsf{read}' : n \mapsto (n, n)$ and $\mathsf{write}'_m : n \mapsto (*, m)$.

In [2], it was observed that interaction laws could be used to merge algebra with coalgebras, constructing an algebra describing the whole system. In particular:

Definition 2 ([2])**.** Given a residual monad-comonad interaction law $\phi_{X,Y} : T_S X \times D_P Y \to T_R(X \times Y)$, we can merge a monad algebra $\alpha : T_R A \to A$ with a comonad coalgebra $\beta : B \to D_P B$, getting a monad algebra $m_\phi(\alpha, \beta) : T_S(A^B) \to A^B$.

The carrier contains A-valued predicates on the state space B, hence it can lift A-valued predicates on $X \times B$ to A-valued predicates on $T_S X \times B$. In a sense, given some post-condition $X \times B \to A$, the resulting algebra computes a *weakest precondition* $T_S X \times B \to A$.

As a simple example, given the empty signature $\emptyset$, the free monad $T_\emptyset$ is the identity monad. We have only one monad algebra $id : T_\emptyset \mathbb{B} \to \mathbb{B}$ over the Booleans $\mathbb{B} = \{\top, \bot\}$, which is given by the identity function. Using the interaction law $\mathsf{d}^S_{X,Y} : T_S X \times D_S Y \to X \times Y = T_\emptyset(X \times Y)$, we can transform any coalgebra $\beta : B \to D_S B$ into an algebra $m_{\mathsf{d}^S}(id, \beta) : T_S(\mathbb{B}^B) \to \mathbb{B}^B$. Here, $\mathbb{B}^B$ is isomorphic to the powerset $\mathcal{P}(B)$, and can be seen as the set of predicates for B.

More nuanced examples can be constructed using local definitions of algebras and coalgebras, and monad morphisms to model the interaction laws. For instance we have two examples:

Costly interactions: Using the time taken algebra on $\mathbb{N} \times X$ from the time example, we can add a notion of cost to any interaction with an environment, for instance tracking the number of times the state of a global store is accessed.

Probabilistic responses: Using the probability algebra, we can model environments which give probabilistically weighted answers, e.g. given by streams of distributions.

The above gives us a way to wrap up the behaviour of entire networks of program-environment interactions into a single algebra, which allows us to efficiently describe relevant verification properties for the components of the network, including the programs that generate interaction laws. Moreover, with tools from domain theory, we can model general recursive programs too, which with the results from [3] gives rise to notions of behavioural equivalence.

In this talk, we will delve into the theory behind these descriptions, and go over a variety of examples of algebraic models for program environment interactions, both in their abstract categorical form and using explicit program codes for generating them.

References

[1] Shin-ya Katsumata, Exequiel Rivas, and Tarmo Uustalu. Interaction laws of monads and comonads. 2020. Thirty-Fifth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS).

[2] Tarmo Uustalu and Niels F. W. Voorneveld. Algebraic and coalgebraic perspectives on interaction laws. In *Programming Languages and Systems - 18th Asian Symposium, APLAS 2020*, volume 12470 of *Lecture Notes in Computer Science*, pages 186–205. Springer, 2020.

[3] Niels Voorneveld. Quantitative logics for equivalence of effectful programs. *Electronic Notes in Theoretical Computer Science*, 347:281 – 301, 2019. Proceedings of the Thirty-Fifth Conference on the Mathematical Foundations of Programming Semantics.