Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 298 (2013) 309–324

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Monads for Behaviour

Maciej Piróg[1] Jeremy Gibbons[2]

Department of Computer Science
University of Oxford

Abstract

The monads used to model effectful computations traditionally concentrate on the 'destination'—the final results of the program. However, sometimes we are also interested in the 'journey'—the intermediate course of a computation—especially when reasoning about non-terminating interactive systems. In this article we claim that a necessary property of a monad for it to be able to describe the behaviour of a program is complete iterativity. We show how an ordinary monad can be modified to disclose more about its internal computational behaviour, by applying an associated transformer to a completely iterative monad. To illustrate this, we introduce two new constructions: a coinductive cousin of Cenciarelli and Moggi's generalised resumption transformer, and States—a State-like monad that accumulates the intermediate states.

Keywords: completely iterative monads, effects, tracing, resumptions

1 Introduction

In this article we are concerned with semantics of programs like the following Haskell fragment:

```
echo :: IO ()
echo = do { x <- getChar ; putChar x ; echo }
```

More precisely, we are interested in programs that (1) have side-effects, and (2) depend on a (not necessarily terminating) recursion—or a corecursion, if you will. In the example, `echo` performs observable actions and then calls itself, 'unfolding' an infinite series of events.

Since Moggi's work [24], monads have become the standard model for computational effects. For example, a popular choice to model I/O operations is to employ the State monad $A \mapsto (A \times S)^S$, model the outside world as an object S, and see the program semantics as a function transforming an initial state into a final

[1] maciej.pirog@cs.ox.ac.uk
[2] jeremy.gibbons@cs.ox.ac.uk

http://dx.doi.org/10.1016/j.entcs.2013.09.019

state [7,18]. Alternatively, we could consider side-effects as communication with the environment, so no assumption about semantics of effects needs to be made at this point: the program semantics is a free structure generated by the 'effectful' constructs (`getChar` and `putChar`), which is then interpreted by an external handler [16,29,32].

The situation becomes much more complicated in the context of (2). While a pure corecursive function is often denoted by the unique homomorphism to the final coalgebra that represents the type of the unfolded codata, a monad that models the effects in question need not be the carrier of a final coalgebra. Instead, we turn our attention to monads that come equipped with their own corecursive structure—*completely iterative monads* ('cims'), introduced by Elgot [12] and recently studied by Aczel *et al.* [1,22]. A monad M is a cim if for certain morphisms $e : X \to M(A + X)$ there exists a unique morphism $e^\dagger : X \to MA$ that is coherent with the monadic structure of M (for the full definition, see Section 2). This rather axiomatic approach makes it possible to separate the corecursion guarded by invocation of effects from any recursive structure enjoyed by the base category, like order or metric enrichment. As an example, in Section 4.1 we give a categorical interpretation of generalised While programs that do not need to terminate.

A known example of a cim is the free cim Σ^∞ generated by an endofunctor Σ. It is given by $\Sigma^\infty A = \nu X.A + \Sigma X$. However, the mentioned State and free monads are not in general completely iterative. For example, the State monad does not build the final state incrementally, so in case of non-terminating programs, such as `echo`, it is useless. The free structure, on the other hand, sometimes needs to be infinite, so in general the free monad Σ^* (for an endofunctor Σ representing the signature) is 'too small'. Nevertheless, we should not discard the 'usual' monads too hastily. For example, if we program a divergent computation in the State monad, the intermediate states are physically 'put' and 'gotten' somewhere in the memory of the computer, so the internal behaviour of the computation is, in a sense, accurate. The point is to reify it as a mathematical model. An interesting fact is that the `IO` monad in the Haskell Glasgow Compiler (GHC) is implemented using the State monad [20], so whatever its mathematical model, the two presumably have to be related.

Our idea is to use transformers associated with the 'usual' monads to trace computations. For a cim T and an adjunction $F \dashv U$ that gives rise to a monad M (that is, $UF = M$), we use the monad UTF to trace computations in M. Clearly, UTF supports M-computations (via the canonical monad morphism $M \to UTF$), but it can also store some observations about the course of the computation in the inner cim. The choice of the monad T and the adjunction reveals different aspects of computations in M. As our main technical result, we prove that UTF is completely iterative.

As an example, we use the currying adjunction to derive what we call the States monad, which behaves like State, but also gathers the intermediate states in a stream. This way, the result of the computation is not a single, final state, but rather a possibly infinite trace consisting of intermediate states.

We then introduce the Coinductive Generalised Resumption transformer $M(\Sigma M)^\infty$, which is a coalgebraic cousin of Cenciarelli and Moggi's Generalised Resumption transformer $M(\Sigma M)^*$ [9]. It allows one to decompose a monadic computation into a possibly infinite number of steps interleaved with free structure. It is also a categorical model for datatypes built around resumptions, such as Haskell iteratees [21] (for $\Sigma A = A^{1+I}$) or pipes [15] (for $\Sigma A = A^I + A \times O$), used to perform resource-aware lazy I/O. The fact that we use the free cim is crucial, since programming patterns for iteratees and pipes rely heavily on coalgebraic computations.

2 Completely iterative monads

2.1 Initial assumptions and notations

For the entire article, we assume that we are working in a base category $\mathcal{B}$ with binary coproducts and all the necessary final coalgebras. We denote the coproduct injections by inl and inr. We use a subscript for the composition of a natural transformation with a functor; for example, for functors H and J, if $\xi : F \to G$ is natural, then $\xi_H : FH \to GH$. If ξ is natural in two variables, by $\xi_{H,J}$ we mean a natural transformation ζ with components $\zeta_A = \xi_{HA,JA}$.

We also recall the standard interpretation of coinductive datatypes as final coalgebras. For an endofunctor F, an F-coalgebra is a pair $\langle A, f : A \to FA\rangle$. We call A the *carrier* of the coalgebra. A morphism $h : A \to B$ is an F-coalgebra *homomorphism*, denoted as $h : \langle A, f\rangle \to \langle B, g\rangle$, if $g \cdot h = Fh \cdot f$. An F-coalgebra $\langle \nu F, \beta\rangle$ is *final* if for every F-coalgebra $\langle A, f\rangle$ there exists precisely one homomorphism $\langle A, f\rangle \to \langle \nu F, \beta\rangle$, called an *anamorphism* and denoted as $[\![f]\!]$.

2.2 Cims defined

An anamorphism allows to unfold a (possibly infinite) data structure, while its uniqueness amounts to the principle of coinduction. A corecursive monadic computation can also be described by a coalgebra $e : X \to M(A + X)$, called an *equation morphism*. The object X represents (a set of) *variables*—the seeds of the corecursion. The object A represents (a set of) *parameters*, which are final values of the computation. However, the described computation is not intended to be unfolded in the final $M(A + \text{-})$-coalgebra. Instead, the results of subsequent steps are combined using the monad multiplication. Of course, such (possibly infinite) multiplications need not exist for a monad. Their existence (and uniqueness) is the defining property of completely iterative monads.

We note that not every equation morphism describes a meaningful computation. For example, a morphism that incessantly returns the seed with the unit of the monad is intuitively a pure divergent computation, which does not have an interpretation in the category Set. Thus, we assume the view that some computations generate observable behaviour of the program, while others are 'silent'. We restrict equation morphisms to those that always generate observable actions. We call such morphisms *guarded*. It guarantees that each step of the computation

contributes a new bit of observable behaviour. Hence, following the type theoretic nomenclature [10], we call such a computation *productive*.

To formalise this, we need the notion of *ideals* of a monad. Analogously to ideals in a ring or a semigroup (subsets closed under the operations), they mark a subset of effects encompassed by the monad, for example the subset of observable actions. Informally, once an action from an ideal is performed, it cannot be undone. More precisely, a computation in the ideal composed with any other operation is again in the ideal. An example is non-failing and non-idempotent nondeterminism; it can be idealised with nondeterministic computations with at least n possible results: once n choices are made, there will be at least n (possibly duplicate) final answers, no matter what is the rest of the computation (assuming termination). We can formalise it as follows. (All the definitions in this section are as given by Adámek, Milius, and Velebil [2].)

Definition 2.1 *Let $\langle M, \eta, \mu\rangle$ be a monad. For an endofunctor $\overline{M}$, a natural transformation $\sigma : \overline{M} \to M$ with monomorphic components is called a subfunctor of M. We call σ an ideal of M if there exists a natural transformation $\overline{\mu} : \overline{M}M \to \overline{M}$ such that the following diagram commutes.*

$$\begin{array}{ccc} \overline{M}M & \xrightarrow{\sigma_M} & M^2 \\ \downarrow{\scriptstyle \overline{\mu}} & & \downarrow{\scriptstyle \mu} \\ \overline{M} & \xrightarrow{\sigma} & M \end{array}$$

We call a pair of a monad and its ideal an idealised monad. An idealised monad M is called an ideal monad if $M = \mathsf{Id} + \overline{M}$ with $\eta = \mathsf{inl}_{\mathsf{Id},\overline{M}}$ and $\sigma = \mathsf{inr}_{\mathsf{Id},\overline{M}}$.

Examples of ideal monads include: free monads, exceptions, interactive output, and nonempty lists. Note that in a category with an initial object 0, every monad M is idealised with respect to the trivial ideal $FX = 0$, that is a constant functor that always returns the initial object.

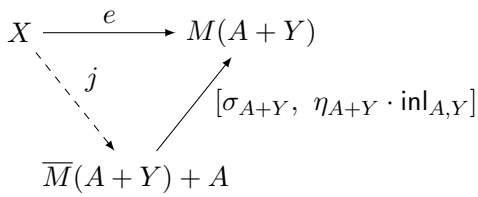
We also need morphisms that respect the internal structure of idealised monads. If Σ is an endofunctor, then a natural transformation $\xi : \Sigma \to M$ is ideal if its codomain contains only observable computations. Intuitively, this means that an interpretation of a symbol from the signature should never yield a silent computation. Formally:

Definition 2.2 *Let $\langle M, \sigma^M\rangle$ and $\langle N, \sigma^N\rangle$ be idealised monads. A natural transformation $\xi : \Sigma \to M$ is ideal if it factors through σ^M.*

A monad is completely iterative *with respect to* an ideal, which contains the observable, corecursive effects of the monad. Thus, we restrict codomains of equation morphisms to ideals. This makes the corecursion guarded by invocation of observable effects.

Definition 2.3 *A morphism $e : X \to M(A + Y)$ is guarded if it factors through the morphism $[\sigma_{A+Y}, \eta_{A+Y} \cdot \mathsf{inl}_{A,Y}]$, that is there exists a morphism j such that the*

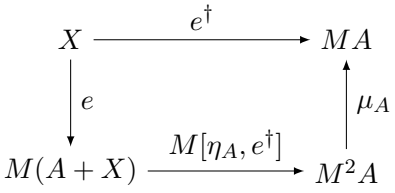
following diagram commutes.



If $X = Y$, we call e a guarded equation morphism.

We use a guarded equation morphism e to unfold a computation $e^\dagger$, called a solution. Intuitively, a solution is an infinite iteration of parameter-preserving Kleisli-compositions of e. A monad is a cim if such a composition always exists and is unique. Formally:

Definition 2.4 *Let $e : X \to M(A + X)$ be a morphism. We call a morphism $e^\dagger : X \to MA$ a solution of e if the following diagram commutes.*



An idealised monad M is completely iterative if every guarded equation morphism has a unique solution.

2.3 The free cim

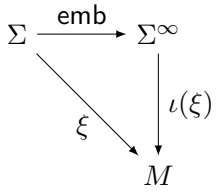
An example of a cim is a generalisation of the infinite term monad generated by an endofunctor (intuitively, a signature) Σ. Its functorial part is given by a family of final coalgebras $\Sigma^\infty A = \nu X.A + \Sigma X$. Below we define the unit η^∞ and a natural transformation $\mathsf{emb} : \Sigma \to \Sigma^\infty$ that embeds Σ in Σ^∞.

$$\begin{array}{c} \mathsf{Id} \\ \downarrow \eta^\infty = \mathsf{inl}_{\mathsf{Id},\Sigma\Sigma^\infty} \\ \mathsf{Id} + \Sigma\Sigma^\infty \cong \Sigma^\infty \end{array} \qquad \begin{array}{c} \Sigma \\ \downarrow \mathsf{emb} = \mathsf{inr}_{\mathsf{Id},\Sigma\Sigma^\infty} \cdot \Sigma\eta^\infty \\ \mathsf{Id} + \Sigma\Sigma^\infty \cong \Sigma^\infty \end{array}$$

The multiplication μ^∞ can be described with the following universal property: it is a unique morphism $u : \Sigma^\infty\Sigma^\infty \to \Sigma^\infty$ that satisfies the following equation.

$$\begin{array}{rl} u = & \Sigma^\infty\Sigma^\infty \cong \Sigma^\infty + \Sigma\Sigma^\infty\Sigma^\infty \cong \mathsf{Id} + \Sigma\Sigma^\infty + \Sigma\Sigma^\infty\Sigma^\infty \\ & \qquad\qquad\qquad\qquad\qquad\qquad \downarrow \mathsf{id} + [\mathsf{id}, \Sigma u] \\ & \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{Id} + \Sigma\Sigma^\infty \cong \Sigma^\infty \end{array}$$

As discussed by Aczel *et al.* [1], Σ^∞ is the free cim generated by Σ. Intuitively, this means that every interpretation of Σ in a cim M extends in a unique way to an interpretation of the entire (possibly infinite) term Σ^∞ in M. Formally, for an ideal natural transformation $\xi : \Sigma \to M$, there exists a unique monad morphism $\iota(\xi) : \Sigma^\infty \to M$ such that the following diagram commutes.



The monad morphism $\iota(\xi)$ is given by $[\eta^M, \xi^\dagger_{\Sigma^\infty}]$. Diagrammatically:

$$\begin{array}{c} \Sigma\Sigma^\infty \\ \downarrow{\scriptstyle \xi_{\Sigma^\infty}} \\ M\Sigma^\infty \cong M(\mathsf{Id} + \Sigma\Sigma^\infty) \end{array} \qquad \begin{array}{c} \Sigma\Sigma^\infty \\ \downarrow{\scriptstyle \xi^\dagger_{\Sigma^\infty}} \\ M\mathsf{Id} = M \end{array} \qquad \begin{array}{c} \Sigma^\infty \cong \mathsf{Id} + \Sigma\Sigma^\infty \\ \downarrow{\scriptstyle \iota(\xi) = [\eta^M, \xi^\dagger_{\Sigma^\infty}]} \\ M \end{array}$$

Another example of a cim is the Exception monad $A \mapsto A + E$. Also, every monad is completely iterative with respect to the trivial ideal $FX = 0$. But, except for those and the free cim, there are hardly any examples of cims commonly used in programming or semantics. This paper aims to fill this void in a rather generic fashion.

3 Cims, adjunctions, and tracing

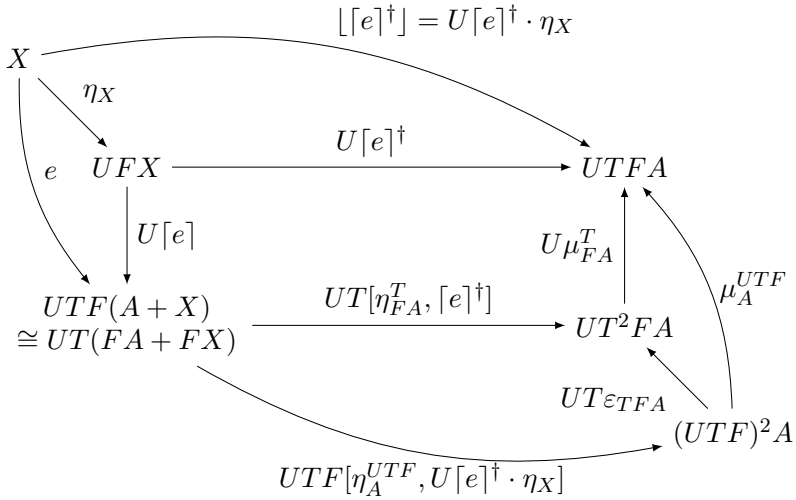
Let M be a monad, and let $\langle F, U, \eta, \varepsilon\rangle : \mathcal{B} \to \mathcal{C}$ be a factorization of M as an adjunction, that is $M = \langle UF, \eta, U\varepsilon_F\rangle$. Let $\langle T, \eta^T, \mu^T, \sigma^T\rangle$ be a cim with solutions $\text{-}^\dagger$. It is standard that UTF is a monad with $\eta^{UTF} = U\eta^T_F \cdot \eta$ and $\mu^{UTF} = U\mu^T_F \cdot UT\varepsilon_{TF}$, and that $\mathsf{lift} = U\eta^T_F : UF \to UTF$ is a monad morphism. We prove that UTF inherits complete iterativity from T.

Theorem 3.1 *The natural transformation $U\sigma^T_F : U\overline{T}F \to UTF$ forms an ideal. The monad UTF is completely iterative with respect to this ideal.*

Proof. Right adjoints preserve monomorphisms, hence the components of the natural transformation $U\sigma^T_F$ are monic, and so it is a subfunctor. We define $\overline{\mu}$ to be $U\overline{\mu^T_F} \cdot U\overline{T}\varepsilon_{TF}$. It is easy to verify that it satisfies the condition for ideals.

Let $e : X \to UTF(A + X)$ be a $U\sigma^T_F$-guarded equation morphism. By $\lfloor\text{-}\rfloor : \mathcal{C}[FA, B] \cong \mathcal{B}[A, UB] : \lceil\text{-}\rceil$ we denote the natural isomorphism associated with the adjunction. Recall that left adjoints preserve coproducts, that is $F(A+B) \cong FA + FB$. One can calculate that $\lceil e\rceil \cong [\sigma^T_{(FA+FX)}, \eta^T_{(FA+FX)} \cdot \mathsf{inl}_{(FA,FX)}] \cdot (\varepsilon_{\overline{T}F(A+X)} + \mathsf{id}_{FA}) \cdot Fj$, which means that $\lceil e\rceil : FX \to TF(A + X) \cong T(FA + FX)$ is a guarded equation morphism in T with a unique solution $\lceil e\rceil^\dagger : FX \to TFA$.

We define the solution of e as $\lfloor \lceil e \rceil^\dagger \rfloor$. The following diagram commutes:



The inner square is the U-image of the solution diagram for $\lceil e \rceil^\dagger$. The outer triangles commute due to properties of adjunctions and the definition of μ^{UTF}.

For uniqueness, let $g : X \to UTFA$ be a solution of e. Substitute $\lceil g \rceil$ for $\lceil e \rceil^\dagger$ in the above diagram. The outer square commutes, because $\lfloor \lceil g \rceil \rfloor = g$ is a solution, and the triangles commute, because of properties of adjunctions, hence the inner square precomposed with η_X also commutes. For all morphisms $f, f' : FB \to C$, if $Uf \cdot \eta_B = Uf' \cdot \eta_B$ then $f = f'$. Therefore, $\lceil g \rceil$ is a solution of $\lceil e \rceil$, so $\lceil g \rceil = \lceil e \rceil^\dagger$, hence $g = \lfloor \lceil g \rceil \rfloor = \lfloor \lceil e \rceil^\dagger \rfloor$. □

Intuitively, T collects observations about a computation in M. Thus, we need a new operation that allows us to actually observe the current state of the computation, for example the current state in the State monad (this example is elaborated in the next section). It could be given as a natural transformation $\mathsf{olift} : M \to UTF$ with components that factor through $U\sigma^T_F$. It will not in general be a monad morphism; on the contrary, performing two actions and then observing the effect differs in general from observing the effect of each action individually. More formally, let $f \circ g$ be a computation in the Kleisli category of M, where $\circ$ is the Kleisli composition. We can decorate it with observers in two different ways: $\mathsf{olift} \cdot (f \circ g)$ or $(\mathsf{olift} \cdot f) \circ (\mathsf{olift} \cdot g)$. For example, when tracing a computation in State, we may want to observe only 'put' operations, as long as we are certain that there are only finitely many invocations of 'get' in between every two invocations of 'put'. In the rest of the paper we always define olift as $U\mathsf{obs}$ for a natural transformation $\mathsf{obs} : F \to TF$. For convenience, we also define a 'save the current state of computation' operation $\mathsf{save} = \mathsf{olift} \cdot \eta : \mathsf{Id} \to UTF$.

Though we do not use this property directly in the rest of the article, observations should not modify the computation. This could be captured by the following cancellation property: for all morphisms $f, f' : A \to MB$ and $g, g' : B \to MC$, if $(\mathsf{lift} \cdot g) \circ \mathsf{save}_B \circ (\mathsf{lift} \cdot f) = (\mathsf{lift} \cdot g') \circ \mathsf{save}_B \circ (\mathsf{lift} \cdot f')$ then $g \circ f = g' \circ f'$.

4 The States monad

Our first example is a monad we call States. If the base category $\mathcal{B}$ is cartesian closed, the State monad arises from the currying adjunction $- \times S \dashv -^S$. We choose $(- \times S)^\infty$, for which we write $\overrightarrow{S}$, to be the inner cim, and the result is the monad $A \mapsto (\overrightarrow{S}(A \times S))^S$. Intuitively, $\overrightarrow{S}$ is a possibly infinite stream of states of type S. The 'base' of the exponential is the trace of the computation: a stream that, if finite, is terminated with an answer A and a current state S. The latter is used only to compose two computations and is not stored in the stream.

We define 'put' and 'get' operations as standard liftings of 'put' and 'get' for State. The natural transformation obs duplicates the current state and puts it in the stream as follows, where $\mathsf{outl}_{A,B} : A \times B \to A$ and $\mathsf{outr}_{A,B} : A \times B \to B$ are the left and right projections respectively.

$$A \times S \xrightarrow{\langle\langle \mathsf{outl}, \mathsf{outr}\rangle, \mathsf{outr}\rangle} (A \times S) \times S \xrightarrow{\mathsf{emb}_{A\times S}} \overrightarrow{S}(A \times S)$$

For example, consider the following computation in States on Set for $S = \mathbb{N}$ (using Haskell syntax):

```
let f = do {put 2; save; put 3; save; put 5}
    g = do {x <- get; put (x+1); save; g}
in do {f; g}
```

For any initial state, `f` evaluates to the trace $(2, 3, \langle\star, 5\rangle)$, while the whole computation evaluates to $(2, 3, 6, 7, 8, 9, \ldots)$.

4.1 Example: Control structures for While

Consider a generalised While language, as given by Rutten [30]:

$$P, Q \ ::= \ A \mid P; Q \mid \textbf{if } b \textbf{ then } P \textbf{ else } Q \mid \textbf{while } b \textbf{ do } P$$

For a monad M, the symbol A represents a set of actions (denoted as $\underline{a}$), that is morphisms of type $1 \to M1$. The symbol b represents elements of a set B of Boolean expressions, that is a set of morphisms of type $1 \to M(1 + 1)$. We parametrise the semantics with a 'guard' operation $\gamma : 1 \to M1$, which allows the addition of behaviour on every choice point of a control structure. The denotation of a program P is given by $[\![P]\!] : 1 \to M1$, defined as follows, where $\circ$ is Kleisli composition.

$$\begin{aligned}
[\![\underline{a}]\!] &= \underline{a} \\
[\![P; Q]\!] &= [\![Q]\!] \circ [\![P]\!] \\
[\![\textbf{if } b \textbf{ then } P \textbf{ else } Q]\!] &= [[\![P]\!], [\![Q]\!]] \circ b \circ \gamma \\
[\![\textbf{while } b \textbf{ do } P]\!] &= ([M\mathsf{inr}_{1,1} \cdot [\![P]\!],\ M\mathsf{inl}_{1,1} \cdot \eta_1^M] \circ b \circ \gamma)^\dagger
\end{aligned}$$

Actions denote themselves, and compositions of programs are just Kleisli compositions of morphisms. The denotation of **if** statements first performs the guard γ, then b, and then the appropriate branch is chosen (we use the left component of $1 + 1$ to represent 'true'). The denotation of **while** first builds an equation morphism by composing the guard, the condition, and the choice between returning the left component of the coproduct (a constant, which means 'stop the iteration'), or performing the body, and right-injecting the result (which makes it a 'continue the iteration' variable). The denotation of the entire **while** expression is a solution to that morphism. The solution might not exist, or might not be unique; hence, depending on the choice of M, A, B, and γ, the denotation might not be well-defined. This semantics specialises to a couple of known cases:

If we choose the regular State monad on Dcppo (the category of pointed directed-complete partial orders and continuous functions) for M and its unit on 1 for γ, the solution diagram simplifies to the familiar equation for denotation of While [27, Chapter 4]. So, if we assume $\text{-}^{\dagger}$ to be the least fixed point, we yield the standard denotational semantics.

If we instantiate M with a cim, we can ensure that unique solutions always exist by an appropriate γ-guarding of **while** loops. (Note that it is not sufficient to ask for the A actions to be guarded, since **while true do while false do** $\underline{a}$ diverges without invoking an action.) In case of the States monad, this means that every iteration stores its initial state in the stream, that is $\gamma = \mathsf{save}$. Additionally, if we assume that 'put' operations are always guarded and 'get' are not, we obtain a semantics trace-equivalent to Nakata and Uustalu's trace operational semantics [26].

5 Coinductive generalised resumptions

Let $\langle M, \eta^M, \mu^M \rangle$ be a monad, and Σ be an endofunctor on the base category $\mathcal{B}$. In this section we give a monadic structure to $M(\Sigma M)^\infty$ and examine its basic properties. We proceed by first giving a monadic structure to the endofunctor

$$KA = \nu X.M(A + \Sigma X),$$

which is isomorphic to $M(\Sigma M)^\infty$ through the coalgebraic version of the rolling rule [5]:

Lemma 5.1 *Let F, G be endofunctors. Then $\nu FG \cong F\nu GF$.*

For convenience, we define two auxiliary natural transformations. The first one, $\mathsf{flat}_{A,B} : M(MA+B) \to M(A+B)$, flattens a computation that may return a value or a new computation. The second one, $\mathsf{unf} : K^2 \to M(\mathsf{Id} + \Sigma K^2)$, unfolds and flattens two levels of structure of K. Note that the final coalgebra map $\alpha_A : KA \to M(A + \Sigma KA)$ is natural in A.

$$\mathsf{flat}_{A,B} = \begin{array}{c} M(MA+B) \\ \downarrow M(\mathsf{id}_{MA} + \eta^M_B) \\ M(MA+MB) \\ \downarrow M[M\mathsf{inl}_{A,B}, M\mathsf{inr}_{A,B}] \\ M^2(A+B) \\ \downarrow \mu^M_{A+B} \\ M(A+B) \end{array} \qquad \mathsf{unf} = \begin{array}{c} K^2 \\ \downarrow \alpha_K \\ M(K+\Sigma K^2) \\ \downarrow M(\alpha + \mathsf{id}_{\Sigma K^2}) \\ M(M(\mathsf{Id}+\Sigma K)+\Sigma K^2) \\ \downarrow \mathsf{flat}_{\mathsf{Id}+\Sigma K, \Sigma K^2} \\ M(\mathsf{Id}+\Sigma K+\Sigma K^2) \end{array}$$

The unit η^K of the monad K is given below. The multiplication is defined as the anamorphism $\mu^K_A = [\![m_A]\!]$ for the following natural transformation m.

$$\eta^K = \begin{array}{c} \mathsf{Id} \\ \downarrow \mathsf{inl}_{\mathsf{Id},\Sigma K} \\ \mathsf{Id}+\Sigma K \\ \downarrow \eta^M_{\mathsf{Id}+\Sigma K} \\ M(\mathsf{Id}+\Sigma K) \cong K \end{array} \qquad m = \begin{array}{c} K^2 \\ \downarrow \mathsf{unf} \\ M(\mathsf{Id}+\Sigma K+\Sigma K^2) \\ \downarrow M(\mathsf{id} + [\Sigma\eta^K_K, \mathsf{id}_{\Sigma K^2}]) \\ M(\mathsf{Id}+\Sigma K^2) \end{array}$$

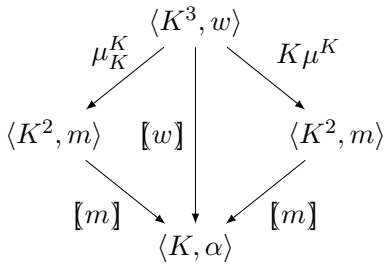
Theorem 5.2 *The following hold:*

(i) *The tuple* $\langle K, \eta^K, \mu^K\rangle$ *is a monad,*

(ii) *There exists a monad distributive law* $\lambda : (\Sigma M)^\infty M \to M(\Sigma M)^\infty$ *given by* $\lambda = \mu^K \cdot M(\Sigma M)^\infty M\eta^\infty \cdot \eta^M_{(\Sigma M)^\infty M}$,

(iii) *There exist two monad morphisms* $\mathsf{liftl} : M \to M(\Sigma M)^\infty$ *and* $\mathsf{liftr} : \Sigma^\infty \to M(\Sigma M)^\infty$.

Proof. The statement (i) can be proved by the structural coinduction provided by the finality of K. For example, to prove the associativity of the monad multiplication, one can define a natural transformation $w : K^3 \to M(\mathsf{Id} + \Sigma K^3)$ in a way similar to the transformation m, and calculate that $m \cdot \mu^K_K = M(\mathsf{id} + \Sigma\mu^K_K) \cdot w$ and $m \cdot K\mu^K = M(\mathsf{id} + \Sigma K\mu^K) \cdot w$, which means that both μ^K_K and $K\mu^K$ are coalgebra homomorphisms $\mu^K_K, K\mu^K : \langle K^3, w\rangle \to \langle K^2, m\rangle$. By uniqueness, $\mu^K \cdot \mu^K_K = [\![m]\!] \cdot \mu^K_K = [\![w]\!] = [\![m]\!] \cdot K\mu^K = \mu^K \cdot K\mu^K$. Diagrammatically:



The distributive law λ can be obtained from Barr and Wells' notion of compatibility of monads [6, Chapter 9], which in this case amounts to the following equalities (note that Barr and Wells' book give five conditions for compatibility, but the last two are redundant, and follow from the first three; see [11] for discussion):

- $\eta^K = \eta^M_{(\Sigma M)^\infty} \cdot \eta^\infty = M\eta^\infty \cdot \eta^M$
- $M\mu^\infty = \mu^K \cdot M(\Sigma M)^\infty \eta^M_{(\Sigma M)^\infty}$
- $\mu^M_{(\Sigma M)^\infty} = \mu^K \cdot M\eta^\infty_{M(\Sigma M)^\infty}$

They can also be proved by the coinduction. The distributive law induces two canonical monad morphisms $M \to K$ and $(\Sigma M)^\infty \to K$. We compose the latter with a monad morphism $\Sigma^\infty \to (\Sigma M)^\infty$ given by $\iota(\mathsf{emb} \cdot \Sigma\eta^M)$. □

Alternatively, the definition of μ^K can be given with the following universal property:

Lemma 5.3 *for any natural transformation $u : K^2 \to K$, we define the transformation $\widetilde{u}$ to be the following composition.*

$$\begin{array}{c} K^2 \\ \downarrow \mathsf{unf} \\ M(\mathsf{Id} + \Sigma K + \Sigma K^2) \\ \downarrow M(\mathsf{id} + [\mathsf{id}_{\Sigma K}, \Sigma u]) \\ M(\mathsf{Id} + \Sigma K) \\ \downarrow \alpha^{-1} \\ K \end{array}$$

Then, $u = \widetilde{u}$ if and only if $u = \mu^K$.

Despite the existence of the cospan $M \to M(\Sigma M)^\infty \leftarrow \Sigma^\infty$, the monad $M(\Sigma M)^\infty$ is in general not a coproduct of M and Σ^∞ as monads. To see that, it is sufficient to assume that the base category is Set, M is ideal, and to recall the construction of coproducts of ideal monads by Ghani and Uustalu [14]. In such a setting the coproduct allows only a finite number of interleavings between M and Σ^∞, so it is distinct from K.

5.1 Complete iterativity of K

Consider the category M-Fema of free Eilenberg-Moore M-algebras, that is, algebras where the carrier is of the shape MA, and the action is defined as μ_A^M. We identify an algebra $\langle MA, \mu_A^M \rangle$ with MA, which makes M-Fema a subcategory of $\mathcal{B}$. It is equivalent to the Kleisli category for M. There is a standard free-underlying adjunction $F \dashv U : \mathcal{B} \to M\text{-Fema}$.

As discussed by Mulry [25], liftings of an endofunctor T on $\mathcal{B}$ to M-Fema are in one-to-one correspondence with distributive laws $TM \to MT$. Moreover, a simple calculation shows that if T has a monadic structure and the distributive law respects this structure, the corresponding lifting $\langle T \rangle$ is also a monad. The monad MT induced by the distributive law is equal to the monad $U\langle T \rangle F$.

Now, consider the monad $(\Sigma M)^\infty$. The monad distributive law λ from Theorem 5.2 gives rise to a lifting $\langle (\Sigma M)^\infty \rangle$, defined on objects as $\langle (\Sigma M)^\infty \rangle MA = M(\Sigma M)^\infty A \cong KA$. The following theorem states that the lifting is also a free cim (note that $M\Sigma$ is an endofunctor also over M-Fema):

Theorem 5.4 *The monad $\langle (\Sigma M)^\infty \rangle$ is the free cim generated by $M\Sigma$ in M-Fema. Therefore, it is completely iterative.*

Proof. For a homomorphism $f : MX \to M(A + \Sigma MX)$ in M-Fema, consider the following diagram in the base category. It commutes, because KA is the carrier of the final $M(A + \Sigma -)$-coalgebra.

$$\begin{array}{ccc} KA & \xrightarrow{\alpha_A} & M(A + \Sigma KA) \\ \uparrow{\scriptstyle [\![f]\!]} & & \uparrow{\scriptstyle M(\mathsf{id}_A + \Sigma [\![f]\!])} \\ MX & \xrightarrow{f} & M(A + \Sigma MX) \end{array}$$

It is easy to check that α_A, α_A^{-1} and $M(A+\Sigma[\![f]\!])$ are also homomorphisms (modulo the isomorphism $KA \cong M((\Sigma M)^\infty)A)$, and $[\![f]\!]$ is a homomorphism as a composition of homomorphisms via the computation law: $[\![f]\!] = \alpha_A^{-1} \cdot M(A + \Sigma[\![f]\!]) \cdot f$. This means that this diagram commutes also in M-Fema. One can define coproducts in M-Fema as $MA \oplus MB = M(A + B)$. Expanding the definitions we obtain that the following diagram commutes in M-Fema, where $\alpha_A^\infty : (\Sigma M)^\infty A \to A + \Sigma M(\Sigma M)^\infty A$ is the action of the final $(A + \Sigma M -)$-coalgebra (the morphism $M\alpha^\infty$ is isomorphic to α via the rolling rule). Moreover, $[\![f]\!]$ is unique with this property, since M-Fema is a subcategory of $\mathcal{B}$.

$$\begin{array}{ccc} \langle (\Sigma M)^\infty \rangle MA & \xrightarrow{M\alpha_A^\infty} & \begin{array}{l} M(A + \Sigma \langle (\Sigma M)^\infty \rangle MA) \\ = MA \oplus M\Sigma \langle (\Sigma M)^\infty \rangle MA \end{array} \\ \uparrow{\scriptstyle [\![f]\!]} & & \uparrow{\scriptstyle M(\mathsf{id}_A + \Sigma [\![f]\!]) = \mathsf{id}_{MA} \oplus M\Sigma [\![f]\!]} \\ MX & \xrightarrow{f} & M(A + \Sigma MX) = MA \oplus M\Sigma MX \end{array}$$

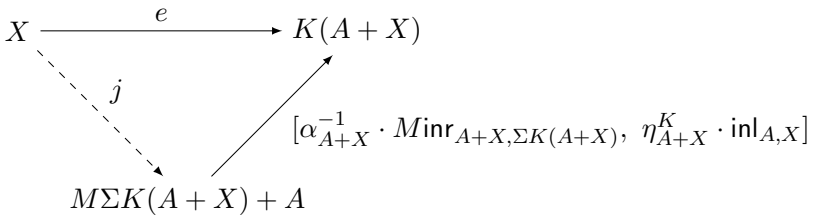
Note that in M-Fema, $M(A+\Sigma-) = MA \oplus M\Sigma-$ is a functor, hence $\langle(\Sigma M)^\infty\rangle MA$ is the carrier of the final $(MA \oplus M\Sigma-)$-coalgebra, and so, according to [22, Corollary 6.3], $\langle(\Sigma M)^\infty\rangle$ is the functorial part of the free cim in M-Fema generated by $M\Sigma$.

It is left to see that the monadic structures given by the lifting and given by the free cim coincide. Here, we show it for multiplications. In case of $\langle(\Sigma M)^\infty\rangle$ the multiplication is given by $M\mu^\infty$. For an object MA, we unfold the universal property of μ^∞ (Section 2.3):

$$
\begin{aligned}
&\langle(\Sigma M)^\infty\rangle\langle(\Sigma M)^\infty\rangle MA = M(\Sigma M)^\infty(\Sigma M)^\infty A \\
&\qquad\qquad \Big\downarrow M(\alpha^\infty + \mathsf{id}) \cdot M\alpha^\infty = (M\alpha^\infty \oplus M\mathsf{id}) \cdot M\alpha^\infty \\
&M(A + \Sigma M(\Sigma M)^\infty A + \Sigma M(\Sigma M)^\infty(\Sigma M)^\infty A) \\
&\qquad\qquad \Big\downarrow M(\mathsf{id} + [\mathsf{id}, \Sigma M\mu^\infty]) = M\mathsf{id} \oplus M[\mathsf{id}, \Sigma M\mu^\infty] \\
&\qquad M(A + \Sigma M(\Sigma M)^\infty A) \\
&\qquad\qquad \Big\downarrow M(\alpha^\infty)^{-1} \\
&\qquad M(\Sigma M)^\infty A = \langle(\Sigma M)^\infty\rangle MA
\end{aligned}
$$

One can show that $M[\mathsf{id}, \Sigma M\mu^\infty]$, where [-,-] is the coproduct mediator in $\mathcal{B}$, is equal to $[[\mathsf{id}_M, M\Sigma M\mu^\infty]]$, where [[-,-]] is the coproduct mediator in M-Fema. Instantiating it in the above composition and some basic properties of functors give us that $M\mu^\infty = (M\alpha^\infty)^{-1} \cdot (\mathsf{id} \oplus [[\mathsf{id}, M\Sigma M\mu^\infty]]) \cdot (M\alpha^\infty \oplus \mathsf{id}) \cdot M\alpha^\infty$, which means that $M\mu^\infty$ satisfies the universal property of the multiplication of the free cim in M-Fema generated by $M\Sigma$. □

The above characterisation and Theorem 3.1 yield that $K \cong U\langle(\Sigma M)^\infty\rangle F$ is completely iterative. The guardedness condition specialises as:



5.2 Example: Bisimulation

Let $\Sigma = \mathsf{Id}$, so that $K \cong MM^\infty$. Similarly to Cenciarelli and Moggi's transformer MM^* [9], a K-computation can be seen as an M-computation split into a series of suspended steps. However, in case of MM^∞, the structure can be infinite, so it can also store a divergent computation. We can see the result of each step as a rather robust observation about the current state of the computation. So, even if the computation does not have a final value, we can still reason about the course of

the computation.

We define the natural transformation $\mathsf{obs} : M \to MM^\infty$ as:

$$M \xrightarrow{M\eta^M} MM \xrightarrow{M\mathsf{emb}} MM^\infty$$

It builds an empty level, so that a composition with another value will not affect the current structure. Intuitively, the outer M is the current state of the computation, while M^∞ is a kind of continuation. To acquire the second state, we can contract the top two steps of execution using a natural transformation force defined as follows, where flat' is equal to flat, but with the monadic argument as the second component of the coproduct rather than the first.

$$\begin{array}{c} MM^\infty \cong M(\mathsf{Id} + MM^\infty) \\ \downarrow \mathsf{flat}'_{\mathsf{Id},M^\infty} \\ M(\mathsf{Id} + M^\infty) \cong M(\mathsf{Id} + \mathsf{Id} + MM^\infty) \\ \downarrow M([\mathsf{id},\mathsf{id}] + \mathsf{id}_{MM^\infty}) \\ M(\mathsf{Id} + MM^\infty) \cong MM^\infty \end{array}$$

On Set, we can define a simple notion of *bisimulation* between programs as a predicate $\approx \subseteq (MM^\infty A)^2$, such that for $p, q \in MM^\infty A$, it is the case that $p \approx q$ precisely if $M(\mathsf{id}_A + !_{M^\infty A})(p) = M(\mathsf{id}_A + !_{M^\infty A})(q)$ and $\mathsf{force}(p) \approx \mathsf{force}(q)$, where $!_A : A \to 1$ is the unique morphism to the final object. In other words, we compare the functorial structure of the outer M (the observable result of the first step), and continue the process after performing the next step with the force natural transformation. This means that two programs are bisimilar if for every $n \in \mathbb{N}$, the respective prefixes of performing the first n steps are equal.

6 Related and future work

Cims arise from completely iterative algebras. Both concepts have been extensively studied by Elgot [12] and by Aczel *et al.* [1,22]. Milius and Moss [23] consider recursive program schemes in terms of solutions in Elgot algebras [3] (that is, Eilenberg-Moore algebras for free cims).

Cenciarelli and Moggi [9] introduced the Generalised Resumption transformer $M(\Sigma M)^*$, which decomposes a monadic computation into a series of steps (layers of free structure). Hyland, Plotkin, and Power [19] proved it to be the coproduct $M + \Sigma^*$ in the category of monads. The monad $M(\Sigma M)^\infty$ captures also potentially infinite computations. In some categories—and so programming languages like Haskell—the limit-colimit coincidence [31] identifies $M(\Sigma M)^*$ and $M(\Sigma M)^\infty$, but the explicit use of the free cim is significant in Set and in type theories with guarded (co)recursion. Interleaving data and monadic actions is a powerful abstraction studied recently also by Filinski and Støvring [13], Atkey *et al.* [4], and the present authors [28].

Since the free cim is a final coalgebra [22], we can see $(M\Sigma)^\infty$ in M-Fema from Theorem 5.4 as an example of Hasuo, Jacobs, and Sokolova's generic trace semantics [17], which models state-based systems as F-coalgebras in a Kleisli category (or, equivalently, a Fema). The coalgebra represents transitions (for example, with $\Sigma A = A \times O$ for labelled transitions), and the monad represents the underlying effect (like the Powerset monad for nondeterminism or the Probability Distribution monad for probabilistic systems).

In this paper we concentrate on the monads and tracing, and we only sketch potential applications in defining semantics and reasoning about programs. The natural next step is to formalise a language like Moggi's computational λ-calculus [24] with recursion provided by a background cim. It is also an interesting question whether the presented theory could be used to develop a practical framework for reasoning about effectful programs in type theories, like those implemented by the Coq or Agda proof systems. So far, Capretta [8] represented general recursion by the free cim generated by the identity functor; we conjecture fruitful applications of other cims too.

Acknowledgment

This work was supported by the UK EPSRC project *Reusability and Dependent Types* (EP/G034516/1). We would like to thank Ralf Hinze, Marek Materzok, Nicolas Wu, and the anonymous reviewers for their comments.

References

[1] Peter Aczel, Jirí Adámek, Stefan Milius, and Jiri Velebil. Infinite trees and completely iterative theories: a coalgebraic view. *Theoretical Computer Science*, 300(1-3):1–45, 2003.

[2] Jirí Adámek, Stefan Milius, and Jiri Velebil. On rational monads and free iterative theories. *Electronic Notes in Theoretical Computer Science*, 69:23–46, 2002.

[3] Jirí Adámek, Stefan Milius, and Jiri Velebil. Elgot algebras. *Logical Methods in Computer Science*, 2(5), 2006.

[4] Robert Atkey, Neil Ghani, Bart Jacobs, and Patricia Johann. Fibrational induction meets effects. In Lars Birkedal, editor, *Foundations of Software Science and Computational Structures—15th International Conference, FoSSaCS 2012*, volume 7213 of *Lecture Notes in Computer Science*, pages 42–57. Springer, 2012.

[5] Roland Carl Backhouse, Marcel Bijsterveld, Rik van Geldrop, and Jaap van der Woude. Categorical fixed point calculus. In David H. Pitt, David E. Rydeheard, and Peter Johnstone, editors, *Category Theory and Computer Science*, volume 953 of *Lecture Notes in Computer Science*, pages 159–179. Springer, 1995.

[6] Michael Barr and Charles F. Wells. *Toposes, Triples, and Theories*. Grundlehren der mathematischen Wissenschaften. Springer-Verlag, 1985.

[7] Andrew Butterfield. Reasoning about I/O in functional programs. In *Proceedings of the 4th Central European Functional Programming School*, CEFP'11, pages 93–141, Berlin, Heidelberg, 2012. Springer-Verlag.

[8] Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2), 2005.

[9] Pietro Cenciarelli and Eugenio Moggi. A syntactic approach to modularity in denotational semantics. In *Proceedings of the 5th Biennial Meeting on Category Theory and Computer Science, CTCS 93, CWI Technical Report*, Amsterdam, The Netherlands, 1993.

[10] Thierry Coquand. Infinite objects in type theory. In Henk Barendregt and Tobias Nipkow, editors, *TYPES*, volume 806 of *Lecture Notes in Computer Science*, pages 62–78. Springer, 1993.

[11] Jeremy E. Dawson. Categories and monads in HOL-Omega. In preparation, http://users.cecs.anu.edu.au/~jeremy/pubs/holw-cm/root.pdf.

[12] Calvin C. Elgot. Monadic computation and iterative algebraic theories. In *Logic Colloquium '73, Proc., Bristol 1973, 175-230*, 1975.

[13] Andrzej Filinski and Kristian Støvring. Inductive reasoning about effectful data types. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 97–110, New York, NY, USA, 2007. ACM.

[14] Neil Ghani and Tarmo Uustalu. Coproducts of ideal monads. *Theoretical Informatics and Applications*, 38(4):321–342, 2004.

[15] Gabriel Gonzalez. The pipes package, 2012. http://hackage.haskell.org/package/pipes.

[16] Peter Hancock and Anton Setzer. Guarded induction and weakly final coalgebras in dependent type theory. In L. Crosilla and P. Schuster, editors, *From Sets and Types to Topology and Analysis. Towards Practicable Foundations for Constructive Mathematics*, pages 115 – 134, Oxford, 2005. Clarendon Press.

[17] Ichiro Hasuo, Bart Jacobs, and Ana Sokolova. Generic trace semantics via coinduction. *Logical Methods in Computer Science*, 3(4), 2007.

[18] Graham Hutton and Diana Fulger. Reasoning about effects: Seeing the wood through the trees. In *Proceedings of the Symposium on Trends in Functional Programming*, Nijmegen, The Netherlands, May 2008.

[19] Martin Hyland, Gordon D. Plotkin, and John Power. Combining effects: Sum and tensor. *Theoretical Computer Science*, 357(1-3):70–99, 2006.

[20] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In Mary S. Van Deusen and Bernard Lang, editors, *Symposium on Principles of Programming Languages, Charleston, South Carolina, USA*, pages 71–84. ACM Press, 1993.

[21] Oleg Kiselyov. Iteratees. In Tom Schrijvers and Peter Thiemann, editors, *FLOPS*, volume 7294 of *Lecture Notes in Computer Science*, pages 166–181. Springer, 2012.

[22] Stefan Milius. Completely iterative algebras and completely iterative monads. *Information and Computation*, 196:1–41, 2005.

[23] Stefan Milius and Lawrence S. Moss. The category-theoretic solution of recursive program schemes. *Theoretical Computer Science*, 366(1-2):3–59, 2006.

[24] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[25] Philip S. Mulry. Lifting theorems for Kleisli categories. In Stephen D. Brookes, Michael G. Main, Austin Melton, Michael W. Mislove, and David A. Schmidt, editors, *Mathematical Foundations of Programming Semantics, 9th International Conference, New Orleans, LA, USA*, volume 802 of *Lecture Notes in Computer Science*, pages 304–319. Springer, 1993.

[26] Keiko Nakata and Tarmo Uustalu. Trace-based coinductive operational semantics for While. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 375–390. Springer, 2009.

[27] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications: a formal introduction*. John Wiley & Sons, Inc., New York, NY, USA, 1992.

[28] Maciej Piróg and Jeremy Gibbons. Tracing monadic computations and representing effects. In James Chapman and Paul Blain Levy, editors, *Proceedings Fourth Workshop on Mathematically Structured Functional Programming, Tallinn, Estonia, 25 March 2012*, volume 76 of *Electronic Proceedings in Theoretical Computer Science*, pages 90–111. Open Publishing Association, 2012.

[29] Gordon D. Plotkin. Adequacy for infinitary algebraic effects (abstract). In *3rd Conference on Algebra and Coalgebra in Computer Science, CALCO 2009, Udine, Italy*, pages 1–2, 2009.

[30] Jan J. M. M. Rutten. A note on coinduction and weak bisimilarity for While programs. *Theoretical Informatics and Applications*, 33(4/5):393–400, 1999.

[31] Michael B. Smyth and Gordon D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal on Computing*, 11(4):761–783, 1982.

[32] Wouter Swierstra and Thorsten Altenkirch. Beauty in the beast: A functional semantics of the awkward squad. In *Haskell '07: Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 25–36, 2007.