Kleene Monads: Handling Iteration in a Framework of Generic Effects

Sergey Goncharov[1], Lutz Schröder[1,2], and Till Mossakowski[1,2] *

[1] Department of Computer Science, University of Bremen
[2] DFKI Laboratory, Bremen

Abstract. Monads are a well-established tool for modelling various computational effects. They form the semantic basis of Moggi's computational metalanguage, the *metalanguage of effects* for short, which made its way into modern functional programming in the shape of Haskell's do-notation. Standard computational idioms call for specific classes of monads that support additional control operations. Here, we introduce Kleene monads, which additionally feature nondeterministic choice and Kleene star, i.e. nondeterministic iteration, and we provide a metalanguage and a sound calculus for Kleene monads, the *metalanguage of control and effects*, which is the natural joint extension of Kleene algebra and the metalanguage of effects. This provides a framework for studying abstract program equality focussing on iteration and effects. These aspects are known to have decidable equational theories when studied in isolation. However, it is well known that decidability breaks easily; e.g. the Horn theory of continuous Kleene algebras fails to be recursively enumerable. Here, we prove several negative results for the metalanguage of control and effects; in particular, already the equational theory of the unrestricted metalanguage of control and effects over continuous Kleene monads fails to be recursively enumerable. We proceed to identify a fragment of this language which still contains both Kleene algebra and the metalanguage of effects and for which the natural axiomatisation is complete, and indeed the equational theory is decidable.

1 Introduction

Program verification by equational reasoning is an attractive concept, as equational reasoning is conceptually simple and in many cases easy to automate. At the same time, equational reasoning is necessarily insufficient for dealing with equality of programs in full Turing-complete programming languages, as the latter will e.g. require induction over data domains; moreover, it is clear that the (observable) equational theory of a Turing-complete programming language fails to be recursively enumerable (it is undecidable by Rice's theorem, and its complement is easily seen to be r.e.) and hence cannot be completely recursively axiomatised. A standard approach is therefore to separate concerns by introducing abstract programming languages that capture only selected aspects of the structure of programs. In such approaches, the abstract level can often be

* Research supported by the DFG project *A construction kit for program logics* (Lu 707/9-1) and by the German Federal Ministry of Education and Research (Project 01 IW 07002 FormalSafe).

handled purely equationally, while reasoning in more complex logics, e.g. first or higher order predicate logics, is encapsulated at lower levels, typically the data level. Two approaches of this kind are Kleene algebra (used here in the version of [9]) and Moggi's monad-based computational metalanguage [12], to which we refer for both brevity and distinctness as the *metalanguage of effects*.

Kleene algebra is essentially the equational logic of regular expressions. Seen as a programming language, Kleene algebra features sequential composition, nondeterministic choice, and iteration in the shape of the Kleene star. When extended with tests, Kleene algebra allows encoding complex control structures including e.g. loops with breaks [11]. Thus, the focus of Kleene algebra as an abstract programming language is the modelling of *nondeterministic control*.

The metalanguage of effects [12] is based on the observation that the bind and return operations constituting a *monad* correspond computationally to sequential composition and promotion of values to computations, respectively, which together with explicit computation types and product types form the basic features of the metalanguage of effects. Besides the fact that the language is higher-order w.r.t. effects in that it allows passing around computations as first-class objects, the chief distinctive feature here is the innocuous-looking return operator, which affords a separation between effectful computations and effectfree values. Thus, the focus of the abstraction is indeed on *effects* in this case, while most control structures including in particular any form of loops are absent in the base language. (Of course, Kleene algebra is also about effectful programs, but has no distinction between effectful and pure functions.)

The metalanguage of effects is sound for a wide range of effects, including e.g. state, exceptions, I/O, resumptions, backtracking, and continuations. Consequently, monads are being used in programming language semantics and in functional programming both to encapsulate side effects and to achieve genericity over side effects. E.g. monads have appeared in an abstract modelling of the Java semantics [6] and in region analysis [13, 3], and they form the basis of functional-imperative programming in Haskell [15]; indeed Haskell's do-notation is essentially the metalanguage of effects.

Here, we study the natural combination of Kleene algebra and the metalanguage of effects, which we call the *metalanguage of control and effects* (MCE). The resulting language is rather expressive; e.g. it supports the following slightly abusive implementation of list reverse. Let $\mathrm{is_empty}$, push, and pop be the usual stack operations, where $\mathrm{is_empty}$ blocks if the state is nonempty, and otherwise does nothing. Then one can define the reverse operation in the MCE as

$$\mathrm{do}\ q \leftarrow (\mathrm{init}\, p \leftarrow \mathrm{ret}\, \mathrm{is_empty}\, \mathrm{in}\, (\mathrm{do}\ x \leftarrow \mathrm{pop}; \mathrm{ret}\ (\mathrm{do}\ p; \mathrm{push}\, x))^*); q$$

The init expression initialises the iteration variable p, and the starred expression is nondeterministically iterated. The program is equivalent to a non-deterministic choice (for all n) of

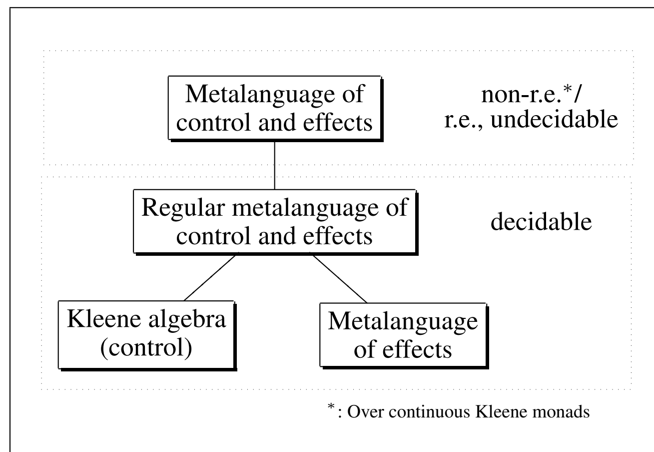
$$\mathrm{do}\ q \leftarrow (\mathrm{do}\ x_1 \leftarrow \mathrm{pop}; \dots; x_n \leftarrow \mathrm{pop}; \mathrm{ret}(\mathrm{do}\ \mathrm{is_empty}; \mathrm{push}\, x_1; \dots; \mathrm{push}\, x_n)); q$$

Hence, this program reads the entire stack into a suspended sequence of push operations and then executes this computation, thus pushing the elements back onto the stack in reversed order. Since the suspended sequence starts with a test for emptiness, it can only be executed if the stack actually is empty — this will discard all sequences of pops not having the length of the stack.

The MCE is interpreted over a class of monads with sufficient additional structure to support iteration, which we call *Kleene monads*; in the same way as in the usual treatment of Kleene algebra, we moreover distinguish *continuous* Kleene monads which interpret the Kleene star as a supremum. We provide a sound calculus for the MCE, which allows for the verification of abstract programs with loops and generic effects. We then proceed to explore the prospects of automatic verification, i.e. we investigate completeness and decidability issues. While Kleene algebra and the metalanguage of effects are equationally complete over the respective natural classes of models [9, 12] and have decidable equational theories (Kleene algebra is even in PSPACE [18]), it is unsurprising in view of our introductory remarks that these properties are sensitive to small extensions of the language; e.g. the Horn theory of continuous Kleene algebras fails to be r.e. [10]. Specifically, we establish the following negative results.

- The equational theory of the MCE over Kleene monads is (r.e. but) undecidable.
- The equational theory of the MCE over continuous Kleene monads is non-r.e. (more precisely at least co-r.e. hard), and hence fails to be finitely axiomatisable.

On the positive side, we show that a natural *regular* fragment of the MCE is complete and decidable. This fragment contains both Kleene algebra and the metalanguage of effects, and hence may be regarded as a suitable abstract framework for combined reasoning about control and effects. The situation is partly summarised in the diagram below.



We illustrate the calculus with an extended example, the proof of a property of a further, more natural implementation of list reverse.

2 Preliminaries: Monads and Generic Imperative Programs

Intuitively, a monad associates to each type A a type TA of computations with results in A; a function with side effects that takes inputs of type A and returns values of type B is, then, just a function of type $A \to TB$. This approach abstracts from particular notions of computation, and a surprisingly large amount of reasoning can be carried out without commitment to a particular type of side effect.

Formally, a *monad* on a category $\mathbf{C}$ can be represented as a *Kleisli triple* $\mathbb{T} = (T, \eta, _^\dagger)$, where $T : \mathrm{Ob}\,\mathbf{C} \to \mathrm{Ob}\,\mathbf{C}$ is a function, the *unit* η is a family of morphisms $\eta_A : A \to TA$, and $_^\dagger$ assigns to each morphism $f : A \to TB$ a morphism $f^\dagger : TA \to TB$ such that

$$\eta_A^\dagger = id_{TA}, \quad f^\dagger \eta_A = f, \quad \text{and} \quad g^\dagger f^\dagger = (g^\dagger f)^\dagger.$$

The *Kleisli category* $\mathbf{C}_{\mathbb{T}}$ of $\mathbb{T}$ has the same objects as $\mathbf{C}$, and $\mathbf{C}$-morphisms $A \to TB$ as morphisms $A \to B$, with *Kleisli composition* $\diamond$ defined by $f \diamond g = f^\dagger \circ g$.

When $\mathbf{C}$ is *Cartesian*, i.e. has finite products, then $\mathbb{T}$ is *strong* if it is equipped with a natural transformation $t_{A,B} : A \times TB \to T(A \times B)$ called *strength*, subject to certain coherence conditions (see e.g. [12]).

Example 1. [12] Computationally relevant strong monads on the category of sets (and, mutatis mutandis, on other categories with sufficient structure) include the following.

1. *Stateful computations with nontermination:* $TA = S \to (1 + A \times S)$, where S is a fixed set of states.
2. *Nondeterminism:* $TA = \mathcal{P}(A)$, where $\mathcal{P}$ is the covariant power set functor.
3. *Exceptions*: $TA = A + E$, where E is a fixed set of exceptions.
4. *Interactive input:* TA is the smallest fixed point of $\gamma \mapsto A + (U \to \gamma)$, where U is a set of input values.
5. *Interactive output:* TA is the smallest fixed point of $\gamma \mapsto A + (U \times \gamma)$, where U is a set of output values.
6. *Nondeterministic stateful computation:* $TA = S \to \mathcal{P}(A \times S)$.

As originally observed by Moggi [12], strong monads support a *computational metalanguage*, i.e. essentially a generic imperative programming language, which we shall refer to as the *metalanguage of effects*. (We consider the first-order version of this language here, which forms the core of [12]; the study of iteration in the computational λ-calculus is the subject of further work. The negative results presented here are in fact made stronger by using a more economic language.) This language is parametrised over a countable signature Σ including a set of atomic types W, from which the type system is generated by the grammar

$$\mathcal{T} ::= W \mid 1 \mid A \times A \mid TA.$$

Moreover, Σ includes basic programs $f : A \to B$ with given profiles, where A and B are types. For purposes of this work, we require that the source type A for f is T-*free*, i.e. does not mention T.

Remark 2. The (trivial) completeness of the MCE over Kleene monads (see below) holds also for arbitrary signatures, but the proof of completeness and decidability of the regular fragment over continuous Kleene monads does depend on the restriction to T-free arguments; this is unsurprising, as basic programs with occurrences of T in their argument types are essentially user-defined control structures. The negative results presented below are actually made stronger by restricting the language.

The terms of the language and their types are then determined by the term formation rules shown in Fig. 1; judgements $\Gamma \rhd t : A$ read 'term t has type A in context Γ', where a *context* is a list $\Gamma = (x_1 : A_1, \ldots, x_n : A_n)$ of typed variables.

$$\textbf{(var)}\ \frac{x : A \in \Gamma}{\Gamma \rhd x : A} \qquad \textbf{(app)}\ \frac{f : A \to B \in \Sigma \quad \Gamma \rhd t : A}{\Gamma \rhd f(t) : B} \qquad \textbf{(1)}\ \frac{}{\Gamma \rhd \langle\rangle : 1}$$

$$\textbf{(pair)}\ \frac{\Gamma \rhd t : A \quad \Gamma \rhd u : B}{\Gamma \rhd \langle t, u\rangle : A \times B} \qquad \textbf{(fst)}\ \frac{\Gamma \rhd t : A \times B}{\Gamma \rhd \mathrm{fst}(t) : A} \qquad \textbf{(snd)}\ \frac{\Gamma \rhd t : A \times B}{\Gamma \rhd \mathrm{snd}(t) : A}$$

$$\textbf{(do)}\ \frac{\Gamma \rhd p : TA \quad \Gamma, x : A \rhd q : TB}{\Gamma \rhd \mathrm{do}\ x \leftarrow p; q : TB} \qquad \textbf{(ret)}\ \frac{\Gamma \rhd t : A}{\Gamma \rhd \mathrm{ret}\, t : TA}$$

Fig. 1. Term formation in the metalanguage of effects

Given an interpretation of atomic types W as objects $[\![W]\!]$ in a Cartesian category $\mathbf{C}$ equipped with a strong monad as above, we obtain obvious interpretations $[\![A]\!]$, $[\![\Gamma]\!]$ of types A and contexts Γ as objects in C. Then an interpretation of basic programs $f : A \to B$ as morphisms $[\![f]\!] : [\![A]\!] \to [\![B]\!]$ induces an interpretation $[\![t]\!] : [\![\Gamma]\!] \to [\![A]\!]$ of terms $\Gamma \rhd t : A$, given by the clauses

$$\begin{aligned}
&[\![x_1 : A_1, \ldots, x_n : A_n \rhd x_i : A_i]\!] = \pi_i^n\\
&[\![\Gamma \rhd f(t) : B]\!] = [\![f]\!] \circ [\![\Gamma \rhd t : A]\!],\ f : A \to B \in \Sigma\\
&[\![\Gamma \rhd \langle\rangle : 1]\!] = !_{[\![\Gamma]\!]}\\
&[\![\Gamma \rhd \langle t, u\rangle : A \times B]\!] = \langle [\![\Gamma \rhd t : A]\!], [\![\Gamma \rhd u : B]\!]\rangle\\
&[\![\Gamma \rhd \mathrm{fst}(t) : A]\!] = \pi_1 \circ [\![\Gamma \rhd t : A \times B]\!]\\
&[\![\Gamma \rhd \mathrm{snd}(t) : A]\!] = \pi_2 \circ [\![\Gamma \rhd t : A \times B]\!]\\
&[\![\Gamma \rhd \mathrm{do}\ x \leftarrow p; q : TB]\!] = [\![\Gamma, x : A \rhd q : TB]\!] \diamond t_{[\![\Gamma]\!],[\![A]\!]} \circ \langle id, [\![\Gamma \rhd p : TA]\!]\rangle\\
&[\![\Gamma \rhd \mathrm{ret}\, t : TA]\!] = \eta_A \circ [\![\Gamma \rhd t : A]\!]
\end{aligned}$$

where the π_i denote projections, $!_A : A \to 1$ is the unique morphism into the terminal object, and $\langle _, _ \rangle$ denotes pairing of morphisms. We say that an *equation* $\Gamma \rhd t = s$, where $\Gamma \rhd t : A$ and $\Gamma \rhd s : A$, is *satisfied* by the interpretation over $\mathbb{T}$, and shortly write $\mathbb{T} \models \Gamma \rhd t = s$, if $[\![\Gamma \rhd t : A]\!] = [\![\Gamma \rhd s : A]\!]$. It is shown in [12] that equality in the metalanguage of effects is completely axiomatised by the standard rules of many-sorted

equational logic plus the equations

$$\begin{aligned}\text{do } x \leftarrow (\text{do } y \leftarrow p; q); r &= \text{do } x \leftarrow p; y \leftarrow q; r \\ \text{do } x \leftarrow \text{ret}\, a; p &= p[a/x] \\ \text{do } x \leftarrow p; \text{ret}\, x &= p\end{aligned}$$

By results of [1], it is moreover immediate that the metalanguage of effects is decidable.

3 Kleene monads

The metalanguage of effects recalled in the previous section is essentially Haskell's do-notation [15]. However, while Haskell's monad libraries include recursively defined loop constructs such as while without further ado, such loops are not provided with reasoning support by the metalanguage of effects: the latter is exclusively a language for *linear sequential programs*. It is the main object of this work to provide an extended generic framework that does support loops. As in classical dynamic logic, we approach this problem from the point of view of regular expressions, i.e. we introduce a nondeterministic iteration construct. We begin with the model for nondeterminism:

Definition 3. An *additive monad* is a monad $\mathbb{T}$ as above, equipped with a factorisation of the hom-functor $\mathsf{Hom}(_,_) : \mathbf{C}_{\mathbb{T}}^{op} \times \mathbf{C}_{\mathbb{T}} \to \mathbf{Set}$ of its Kleisli category through idempotent commutative monoids. If the underlying category $\mathbf{C}$ is Cartesian, this structure is induced by operations $\oplus : TA \times TA \to TA$ and $0 : 1 \to TA$ which satisfy the idempotent commutative monoid laws and distribute over Kleisli composition; we denote the induced structure on the hom-sets in the Kleisli category by $\oplus$ and 0 as well. A *strong additive monad* on a Cartesian category is an additive monad with a tensorial strength t satisfying two extra conditions:

$$\begin{aligned}t_{A,B} \circ \langle id, 0\rangle &= 0 \\ t_{A,B} \circ \langle id, f \oplus g\rangle &= t_{A,B} \circ \langle id, f\rangle \oplus t_{A,B} \circ \langle id, g\rangle.\end{aligned}$$

The operations $\oplus$ and 0 are analogous to the operations mplus and mzero of Haskell's MonadPlus class. Additive monads typically involve powerset. E.g. among the monads listed in Example 1, the powerset monad and the nondeterministic state monad are additive monads. Similarly, nondeterminism may be introduced in other monads; note however that the nondeterministic exception monad $\mathcal{P}(X + E)$ fails to be additive, as it violates distribution of 0 over Kleisli composition (do $x \leftarrow p; 0$ may terminate if p raises an exception) – one way of dealing with this phenomenon is to treat exceptions separately as an outermost layer, in this case on top of the additive monad $\mathcal{P}$ [16]. Alternative axiomatic formulations can be found in [4, 8]. The computational model behind additive monads is *angelic* nondeterminism, i.e. like in nondeterministic Turing machines, nonterminating computation paths are ignored.

We can compare elements of TA by means of the natural ordering: $p \leq q \Leftrightarrow p \oplus q = q$. This turns every $\mathsf{Hom}(X, TA)$ into a join semilattice with least element 0. This leads to the central concept supporting iteration semantically:

Definition 4. An additive monad $\mathbb{T}$ is a *Kleene monad* if the operators

$$f \mapsto p \oplus f \diamond r \quad \text{and} \quad f \mapsto p \oplus r \diamond f$$

(where we agree that $\diamond$ binds more strongly than $\oplus$) both have least fixed points w.r.t. $\langle \mathsf{Hom}(X, TA), \leq \rangle$, and one of two equivalent conditions holds:

$$\mu f.(p \oplus f \diamond r) \diamond q = p \diamond \mu f.(q \oplus r \diamond f) \tag{1}$$

or

$$\mu f.(p \diamond q \oplus f \diamond r) = p \diamond \mu f.(q \oplus f \diamond r) \tag{2}$$

$$\text{and } \mu f.(p \diamond q \oplus r \diamond f) = \mu f.(p \oplus r \diamond f) \diamond q. \tag{3}$$

Intuitively, (1) means that (p followed by a number of r's) followed by q is the same as p followed by (a number of r's followed by q).

A *strong Kleene monad* is a Kleene monad which is strong and whose strength satisfies an extra *strength continuity* condition:

$$\mu f.(t_{A,B} \circ (id \times p) \oplus t_{A,B} \circ (id \times q) \diamond f) = t_{A,B} \circ (id \times \mu f.(p \oplus q \diamond f)),$$

for $p : A \to TB$ and $q : A \to TB$.

Let us show that indeed (1) is equivalent to the conjunction of (2) and (3). Assume (1). Then $\mu f.(p \diamond q \oplus f \diamond r) = p \diamond q \diamond \mu f.(\eta \oplus r \diamond f) = p \diamond \mu f.(q \oplus f \diamond r)$, hence (1) $\Rightarrow$ (2). By a symmetric argument, also (1) $\Rightarrow$ (3).

Now assume (2) and (3). It suffices to prove that $\mu f.(\eta \oplus f \diamond r) = \mu f.(\eta \oplus r \diamond f)$. As $\eta \oplus r \diamond \mu f.(\eta \oplus f \diamond r) = \eta \oplus r \oplus r \diamond \mu f.(\eta \oplus f \diamond r) \diamond r = \eta \oplus (\eta \oplus r \diamond \mu f.(\eta \oplus f \diamond r)) \diamond r$, $\eta \oplus r \diamond \mu f.(\eta \oplus f \diamond r)$ is a fixed point of the map $f \mapsto \eta \oplus f \diamond r$, hence

$$\eta \oplus r \diamond \mu f.(\eta \oplus f \diamond r) \geq \mu f.(\eta \oplus f \diamond r). \tag{4}$$

On the other hand $\mu f.(\eta \oplus f \diamond r) \diamond r = r \oplus \mu f.(\eta \oplus f \diamond r) \diamond r \diamond r$, which means that $\mu f.(\eta \oplus f \diamond r) \diamond r$ is a fixed point of $f \mapsto r \oplus f \diamond r$, hence $\mu f.(\eta \oplus f \diamond r) \diamond r \geq \mu f.(r \oplus f \diamond r)$. Add η to both sides and obtain

$$\mu f.(\eta \oplus f \diamond r) \geq \eta \oplus r \diamond \mu f.(\eta \oplus f \diamond r) \tag{5}$$

From (4) and (5), we have, $\mu f.(\eta \oplus f \diamond r) = \eta \oplus r \diamond \mu f.(\eta \oplus f \diamond r)$, which means that $\mu f.(\eta \oplus f \diamond r)$ is a fixed point of $f \mapsto \eta \oplus r \diamond f$, hence $\mu f.(\eta \oplus f \diamond r) \geq \mu f.(\eta \oplus r \diamond f)$. By symmetric argument $\mu f.(\eta \oplus r \diamond f) \geq \mu f.(\eta \oplus f \diamond r)$ and we are done. □

Equation (1) is a minimal requirement that allows for defining the Kleene star operator correctly. But typically, one deals with Kleene monads satisfying the more restrictive (but natural) condition that $\mu f.(p \oplus f \diamond r) \diamond q = p \diamond \mu f.(q \oplus r \diamond f)$ is the least upper bound of the family of morphisms of the form $p \diamond r \diamond \ldots \diamond r \diamond q$. We call such Kleene monads *ω-continuous*.

Example 5. A sufficient condition for a strong monad to be an ω-continuous Kleene monad is that its Kleisli category be enriched over cpos with finite joins (i.e., countably complete lattices). This holds, e.g., for strong monads over the category of cpos with finite joins, but also for many other examples including all additive monads mentioned above, in particular the nondeterministic state monad in any topos.

On the logical side, we extend the metalanguage of effects to obtain the *metalanguage of control and effects (MCE)* by adding term formation rules

$$\frac{\Gamma \rhd p : TA \quad \Gamma, x : A \rhd q : TA}{\Gamma \rhd \mathrm{init}\, x \leftarrow p \,\mathrm{in}\, q^* : TA} \qquad \frac{\Gamma \rhd p : TA \quad \Gamma \rhd q : TA}{\Gamma \rhd p + q : TA} \qquad \frac{}{\Gamma \rhd \emptyset : TA}$$

While $\emptyset$, $+$ just provide the expected syntax for nondeterminism (deadlock and choice), the $\mathrm{init}\, x \leftarrow p \,\mathrm{in}\, q^*$ construct denotes a nondeterministically chosen number of iterated executions of q, initialised by $x \leftarrow p$, with the result x of the computation fed through the loop. Formally, the new language constructs are interpreted over a strong Kleene monad as follows.

- $[\![\Gamma \rhd \emptyset : TA]\!] = \mathsf{0}$,
- $[\![\Gamma \rhd p + q : TA]\!] = [\![\Gamma \rhd p : TA]\!] \oplus [\![\Gamma \rhd q : TA]\!]$,
- $[\![\Gamma \rhd \mathrm{init}\, x \leftarrow p \,\mathrm{in}\, q^* : TA]\!] =$
  $T\pi_2 \circ \mu f.(t_{[\![\Gamma]\!],[\![A]\!]} \circ \langle id, [\![\Gamma \rhd p : TA]\!]\rangle \oplus t_{[\![\Gamma]\!],[\![A]\!]} \circ \langle \pi_1, [\![\Gamma, x : A \rhd q : TA]\!]\rangle \diamond f)$.

Terms of the Kleene metalanguage will be also referred as *programs*. Programs not involving any monadic constructs (including $\emptyset$) will be called *atomic*. Essentially, atomic programs are combinations of symbols of Σ by means of composition and Cartesian primitives. In Fig. 2 we present an equational axiomatisation MCE, extending the axiomatisation given in the previous section. We leave variable contexts implicit. The order $\leq$ appearing in **(ind$_1$)** and **(ind$_2$)** is the one defined above. The equations **(assoc)**, **(comm)**, and **(idem)** for nondeterministic choice are called the ACI-laws. We refer to the sublanguage of the MCE without $*$ and the corresponding parts of the MCE calculus as the *theory of additive monads*.

Theorem 6 (Soundness). MCE is sound for strong Kleene monads.

Theorem 7 (Completeness). MCE is strongly complete over strong Kleene monads.

We next require some auxiliary machinery for additive monads, which as a side product induces a simple normalisation-based algorithm for deciding equality over additive monads. Consider the following rewriting system, inspired by [2].

$$\begin{array}{rcl}
(p : 1^n) & \rightarrowtail & \langle\rangle^n \\
\langle \mathrm{fst}(p), \langle\rangle^n \rangle & \rightarrowtail & p \\
\langle \langle\rangle^n, \mathrm{snd}(p) \rangle & \rightarrowtail & p \\
\langle \mathrm{fst}(p), \mathrm{snd}(p) \rangle & \rightarrowtail & p \\
\mathrm{do}\ x \leftarrow (p : T1^n); \mathrm{ret}\langle\rangle^n & \rightarrowtail & p \\
\mathrm{do}\ x \leftarrow p; \mathrm{ret}\, x & \rightarrowtail & p
\end{array}
\qquad
\begin{array}{rcl}
\mathrm{fst}\langle p, q\rangle & \rightarrowtail & p \\
\mathrm{snd}\langle p, q\rangle & \rightarrowtail & q \\
\mathrm{do}\ x \leftarrow \mathrm{ret}\, p; q & \rightarrowtail & q[p/x] \quad (*) \\
\mathrm{do}\ x \leftarrow (\mathrm{do}\ y \leftarrow p; q); r & \rightarrowtail & \\
 & & \mathrm{do}\ x \leftarrow p; y \leftarrow q; r
\end{array}$$

Basic monad laws:

$$\textbf{(bind)} \quad \mathrm{do}\ x \leftarrow (\mathrm{do}\ y \leftarrow p; q); r = \mathrm{do}\ x \leftarrow p; y \leftarrow q; r$$
$$\textbf{(eta}_1\textbf{)} \quad \mathrm{do}\ x \leftarrow \mathrm{ret}\, a; p = p[a/x]$$
$$\textbf{(eta}_2\textbf{)} \quad \mathrm{do}\ x \leftarrow p; \mathrm{ret}\, x = p$$

Extra axioms for nondeterminism:

$$\textbf{(plus}\emptyset\textbf{)} \quad p + \emptyset = p \qquad \textbf{(comm)} \quad p + q = q + p$$
$$\textbf{(idem)} \quad p + p = p \qquad \textbf{(assoc)} \quad p + (q + r) = (p + q) + r$$
$$\textbf{(bind}\emptyset_1\textbf{)} \quad \mathrm{do}\ x \leftarrow p; \emptyset = \emptyset \qquad \textbf{(bind}\emptyset_2\textbf{)} \quad \mathrm{do}\ x \leftarrow \emptyset; p = \emptyset$$
$$\textbf{(distr}_1\textbf{)} \quad \mathrm{do}\ x \leftarrow p; (q + r) = \mathrm{do}\ x \leftarrow p; q + \mathrm{do}\ x \leftarrow p; r$$
$$\textbf{(distr}_2\textbf{)} \quad \mathrm{do}\ x \leftarrow (p + q); r = \mathrm{do}\ x \leftarrow p; r + \mathrm{do}\ x \leftarrow q; r$$

Extra axioms and rules for Kleene star:

$$\textbf{(unf}_1\textbf{)} \quad \mathrm{init}\, x \leftarrow p \,\mathrm{in}\, q^* = p + \mathrm{do}\ x \leftarrow (\mathrm{init}\, x \leftarrow p \,\mathrm{in}\, q^*); q$$
$$\textbf{(unf}_2\textbf{)} \quad \mathrm{init}\, x \leftarrow p \,\mathrm{in}\, q^* = p + \mathrm{init}\, x \leftarrow (\mathrm{do}\ x \leftarrow p; q) \,\mathrm{in}\, q^*$$
$$\textbf{(init)} \quad \mathrm{init}\, x \leftarrow (\mathrm{do}\ y \leftarrow p; q) \,\mathrm{in}\, r^* = \mathrm{do}\ y \leftarrow p; \mathrm{init}\, x \leftarrow q \,\mathrm{in}\, r^* \quad (y \notin FV(r))$$

$$\textbf{(ind}_1\textbf{)} \ \frac{\mathrm{do}\ x \leftarrow p; q \leq p}{\mathrm{init}\, x \leftarrow p \,\mathrm{in}\, q^* \leq p} \qquad \textbf{(ind}_2\textbf{)} \ \frac{\mathrm{do}\ x \leftarrow q; r \leq r}{\mathrm{do}\ x \leftarrow (\mathrm{init}\, x \leftarrow p \,\mathrm{in}\, q^*); r \leq \mathrm{do}\ x \leftarrow p; r}$$

Fig. 2. MCE: A calculus for Kleene monads

The rules $(*)$ capture the usual monad laws. Here n ranges over naturals (excluding 0), 1^n denotes the n-fold product of the terminal type 1 with itself, $\langle\rangle^n$ is a shortcut for the n-tuple $\langle\langle\rangle, \ldots, \langle\rangle\rangle$ for $n > 1$ and is just $\langle\rangle$ for $n = 1$.
As usual, we have omitted contexts, which are easily reconstructed except possibly in the second and third rules on the left. Their uncut versions look as follows.

$$\Gamma \rhd \langle \mathrm{fst}(p), \langle\rangle^n \rangle : A \times 1^n \rightarrowtail \Gamma \rhd p : A \times 1^n$$
$$\Gamma \rhd \langle \langle\rangle^n, \mathrm{snd}(p) \rangle : 1^n \times A \rightarrowtail \Gamma \rhd p : 1^n \times A$$

Consider the extra rewrite rules, capturing nondeterminism.

$$\begin{array}{rclrcl} p + \emptyset & \rightarrowtail & p & \mathrm{do}\ x \leftarrow p; \emptyset & \rightarrowtail & \emptyset \\ \emptyset + p & \rightarrowtail & p & \mathrm{do}\ x \leftarrow \emptyset; p & \rightarrowtail & \emptyset \end{array}$$
$$\begin{array}{rcl} \mathrm{do}\ x \leftarrow p; (q + r) & \rightarrowtail & \mathrm{do}\ x \leftarrow p; q + \mathrm{do}\ x \leftarrow p; r \\ \mathrm{do}\ x \leftarrow (p + q); r & \rightarrowtail & \mathrm{do}\ x \leftarrow p; r + \mathrm{do}\ x \leftarrow q; r \end{array} \qquad (**)$$

Let $\rightarrowtail_\lambda$ stand for the reduction relation defined by rules in $(*)$, and $\rightarrowtail_\omega$ for the reduction relation corresponding to the rules in $(**)$. With a slight abuse of notation use $\rightarrowtail$ to refer to the combined relation $\rightarrowtail_\lambda \cup \rightarrowtail_\omega$.

Lemma 8. *The rewrite relation $\rightarrowtail$ is confluent and strongly normalising.*

Denote the normal form of a term t under $\rightarrowtail$ by $\mathsf{nf}(t)$. Note that by definition, the normal form $\mathsf{nf}(t)$ of a program t under $\rightarrowtail$ must always look like $\sum_{i=1}^{n} \mathrm{do}\ x_1^i \leftarrow t_{k_i}^i; \ldots; x_{k_i}^i \leftarrow t_{k_i}^i; r^i$, where the topmost construct in the t_j^i and the r^i is either ret or init or an atomic program. As a simple corollary of Lemma 8 we obtain

Theorem 9. *Two programs p and q are provably equal in the theory of additive monads iff $\mathsf{nf}(p)$ and $\mathsf{nf}(q)$ are equivalent modulo the* ACI-laws.

Proof. The required property is local commutativity with ACI. That is, once p and q are related by an ACI law and $p \rightarrowtail r$, there must exist s such that $q \rightarrowtail^{+} s$, and s is ACI-equivalent to r. This is easily verified by a case analysis. Together with confluence and strong normalisation, this implies by [7] the Church-Rosser property of $\rightarrowtail$ modulo ACI. As the theory of additive monads is precisely ACI plus the unoriented version of rules, defining $\rightarrowtail$, the last property is equivalent to the claim of the lemma. $\square$

Introduce one more (nonterminating) reduction rule for unfolding the Kleene star:

k-rule: $\quad \mathrm{init}\, x \leftarrow p \,\mathrm{in}\, q^* \;\rightarrowtail\; p + \mathrm{init}\, x \leftarrow (\mathrm{do}\ x \leftarrow p; q) \,\mathrm{in}\, q^*$

Define an operator z taking every program to a set of programs as follows:

- $\mathsf{z}(p) = \{p\}$ if is atomic,
- $\mathsf{z}(\mathrm{do}\ x \leftarrow p; q) = \{\mathrm{do}\ x \leftarrow s; t \mid s \in \mathsf{z}(p), t \in \mathsf{z}(q)\}$,
- $\mathsf{z}(\mathrm{ret}\, p) = \{\mathrm{ret}\, p\}$,
- $\mathsf{z}(\emptyset) = \{\,\}$,
- $\mathsf{z}(p + q) = \mathsf{z}(p) \cup \mathsf{z}(q)$,
- $\mathsf{z}(\mathrm{init}\, x \leftarrow p \,\mathrm{in}\, q^*) = \{\,\}$.

Informally, z just flattens nondeterministic choice on the top level into a set, not going under ret's. Then for every program p let $\mathsf{NF}(p) = \bigcup\{\mathsf{z}(\mathsf{nf}_{\lambda\omega}(q)) \mid p \rightarrowtail_{\lambda\omega k}^{*} q\}$. Now for every normal atomic $a : TA$ and a variable $x : A$ define the *derivative* $\delta_{x\leftarrow a}(t) = \{p \mid \mathrm{do}\ x \leftarrow a; p \in \mathsf{NF}(t)\} \cup \{\mathrm{ret}\, x \mid a \in \mathsf{NF}(t)\}$ and the *residue* $\varepsilon(t) = \{p \mid \mathrm{ret}\, p \in \mathsf{NF}(t)\}$. In order to make the choice of the variable irrelevant we identify the derivatives under α-conversion: $\delta_{x\leftarrow a}(p) = (\delta_{y\leftarrow a}(p))[x/y]$.

Lemma 10. *If the equality $p = q$ holds over (ω-continuous) Kleene monads, then for every a, the sets $\delta_{x\leftarrow a}(p)$ and $\delta_{x\leftarrow a}(q)$ are elementwise equal over (ω-continuous) Kleene monads, and so are $\varepsilon(p)$ and $\varepsilon(q)$.*

The proof is by well-founded induction over proofs. For the continuous case this requires the introduction of an extra (infinitary) law:

$$(\omega)\quad \frac{\forall i\ \mathrm{do}\ x \leftarrow (\mathrm{init}\, x \leftarrow p \,\mathrm{in}\, q^i); r \le t}{\mathrm{do}\ x \leftarrow (\mathrm{init}\, x \leftarrow p \,\mathrm{in}\, q^*); r \le t}$$

Here, $\mathrm{init}\, x \leftarrow p \,\mathrm{in}\, q^0 = p$, and
$\mathrm{init}\, x \leftarrow p \,\mathrm{in}\, q^{n+1} = \mathrm{init}\, x \leftarrow (\mathrm{do}\ x \leftarrow p; q) \,\mathrm{in}\, q^n$.

Theorem 11 (Undecidability). *The theory* MCE is undecidable.

Proof. We show the claim by encoding Post's Correspondence Problem (PCP). Let $\Sigma = \{a, b, c\}$, and let $\{\langle p_1, q_1\rangle, \ldots, \langle p_n, q_n\rangle\}$ be an instance of PCP in the alphabet $\Sigma \setminus \{c\}$. We define a program s that generates all possible strings of the form lcr^{-1} with $l = p_{i_1} \ldots p_{i_k}$ and $r = q_{i_1} \ldots q_{i_k}$ for some sequence of indices $i_1, \ldots, i_k$, as follows. If we treat symbols of Σ as morphisms with the typing $1 \to T1$, s can be defined by the term

$$s = \text{do } x \leftarrow \Big(\text{init}\, x \leftarrow \sum_{i=1}^{n} \text{ret}(\text{do } p_i; c; q_i^{-1})\, \text{in} \sum_{i=1}^{n} \text{ret}(\text{do } p_i; x; q_i^{-1})^*\Big); x$$

Then, the fact that the PCP instance does have a solution can be expressed by the inequality

$$\text{ret}\, s \leq \text{do } x \leftarrow r; \text{ret}(s + x) \tag{6}$$

where r is a metaprogram presenting the nondeterministic collection of terms $\text{ret}(\text{do } l; c; l^{-1})$, defined by the term

$$r = \text{init}\, x \leftarrow \text{ret}\, c\, \text{in}\, \text{ret}(\text{do } a; x; a) + \text{ret}(\text{do } b; x; b)^*.$$

By Lemma 10, if the inequality (6) is provable in MCE, then s is provably equivalent to some $s' \in \varepsilon(\text{do } x \leftarrow r; \text{ret}(s+x)) = \{s + \text{do } l; c; l^{-1} \mid l \in \{a, b\}^*\}$. Hence for some l, $\text{do } l; c; l^{-1} \leq s$ must be provable in MCE. By Lemma 10, this is only possible if lcl^{-1} is an element of the set that defines s. On the other hand, if lcl^{-1} is a solution of the PCP, it suffices to apply one of the unfolding axioms finitely many times in order to show that $\text{do } l; c; l^{-1} \leq s$. Likewise, one shows in finitely many steps that $\text{do } l; c; l^{-1} \leq r$. Therefore, we have $\text{do } x \leftarrow r; \text{ret}(s + x) = \text{do } x \leftarrow (r + \text{ret}(\text{do } l; c; l^{-1}); \text{ret}(s + x) = (\text{do } x \leftarrow r; \text{ret}(s + x)) + (\text{do } x \leftarrow \text{ret}(\text{do } l; c; l^{-1}); \text{ret}(s + x)) \geq \text{do } x \leftarrow \text{ret}(\text{do } l; c; l^{-1}); \text{ret}(s + x) = \text{ret}(s + \text{do } l; c; l^{-1}) = \text{ret}\, s$. This inequality is just (6), hence we are done. □

4 Continuity

It might seem useful to take the notion of ω-continuous Kleene monad as the standard definition. Indeed, in this case Kleene star is the least upper bound of its finite approximations, which perfectly matches the intuition. However, we now show that continuous Kleene monads in general are logically intractable. On the other hand, we identify a natural restriction on programs which brings the continuous case back into the realm of tractability.

Theorem 12. *Equality in the MCE over continuous Kleene monads is non-r.e.*

Proof. We prove this theorem in much the same manner as Theorem 11. But instead we encode the dual of the Post's Correspondence Problem, which is co-r.e. complete. This encoding is inspired by [10]. Again, let $\Sigma = \{a, b, c\}$, and $\{\langle p_1, q_1\rangle, \ldots, \langle p_n, q_n\rangle\}$ be an instance of PCP in the alphabet $\Sigma \setminus \{c\}$. Besides the term s from Theorem 11,

which presents all pairs of strings generated by the PCP instance, we need a term t presenting pairs of distinct strings. The program t can be defined, using the term r from Theorem 11, as

$$
\begin{aligned}
t = \mathrm{do}\ x \leftarrow r; y \leftarrow \sharp; z \leftarrow \sharp;\\
u \leftarrow (\mathrm{do}\ y; a; x; (b; z + \mathrm{ret}\langle\rangle) + \mathrm{do}\ y; b; x; (a; z + \mathrm{ret}\langle\rangle) +\\
\mathrm{do}\ (y; a + \mathrm{ret}\langle\rangle); x; b; z + \mathrm{do}\ (y; b + \mathrm{ret}\langle\rangle); x; a; z); u,
\end{aligned}
$$

where $\sharp$ denotes the 'chaos' program

$$
\mathrm{do}\ x \leftarrow (\mathrm{init}\, x \leftarrow \mathrm{ret}\,\mathrm{ret}\langle\rangle\ \mathrm{in}(\mathrm{ret}(\mathrm{do}\ x; a) + \mathrm{ret}(\mathrm{do}\ x; b))^{*}); x.
$$

We are done once we show that the inequality $s \leq t$ holds over all ω-continuous Kleene monads if and only if the PCP instance has no solution. Suppose $s \leq t$ over all ω-continuous monads. In particular, it must be true over the powerset monad $\mathcal{P}$. The interpretation of the terms s and t over $\mathcal{P}$ is precisely as intended, hence the PCP instance cannot have a solution. Now suppose that the PCP instance has no solutions. This means that any string generated by s can also be generated by t. As a consequence of ω-continuity, the interpretations of s and t are precisely the sets of strings they generate. For instance, $[\![s]\!] = \sup_k [\![\mathrm{do}\ x \leftarrow (\mathrm{init}\, x \leftarrow \sum_{i=1}^{n} \mathrm{ret}(\mathrm{do}\ p_i; c; q_i^{-1})\, \mathrm{in} \sum_{i=1}^{n} \mathrm{ret}(\mathrm{do}\ p_i; x; q_i^{-1})^k); x]\!] = \sup_k \sum_{i_1,\ldots,i_k} [\![\mathrm{do}\ p_{i_1}; \ldots; p_{i_k}; c; q_{i_k}^{-1}; \ldots; q_{i_1}^{-1}]\!]$. As the inequality is stable under suprema, s must be greater then t. □

As an immediate consequence of Theorem 12, the calculus MCE is incomplete over ω-continuous Kleene monads. Furthermore, it cannot be completed in any finitary way. Although our encoding of PCP involves only a small part of the expressive power of the language, one may still hope to impose syntactic restrictions on programs in order to regain completeness. One such restriction is to control the use of ret operator, as follows.

Definition 13. The set of *regular programs* is inductively defined by the following rules.

- Any ret-free program is regular,
- All term formation rules, except the rule for Kleene star, generate regular programs from regular premises.

Theorem 14. MCE is complete for equality of regular programs over ω-continuous Kleene monads, and equality of regular programs is decidable.

Proof. Let the equality $p = q$ be valid over all ω-continuous Kleene monads, with p and q regular. We prove the claim by induction over the nesting depth of the ret operator. If it is greater then zero we can decrease it by application of the following routine:

1. In both p and q replace (once) every subterm $\mathrm{init}\, x \leftarrow s\, \mathrm{in}\, t^{*}$, innermost first, by the equivalent term $(s + \mathrm{do}\ x \leftarrow s; \mathrm{init}\, x \leftarrow t\, \mathrm{in}\, t^{*})$, and normalise (by $\rightarrowtail$).

2. In the topmost nondeterministic sum of p and q, replace every atomic t by the equivalent $(\mathrm{do}\ x \leftarrow t; \mathrm{ret}\, x)$.
3. At this point, p and q must take the form $\sum_i \mathrm{do}\ x \leftarrow p_1^i; p_2^i + \sum_j \mathrm{ret}\, p_3^i$ and $\sum_k \mathrm{do}\ x \leftarrow q_1^k; q_2^k + \sum_l \mathrm{ret}\, q_3^l$, respectively, with p_1^i and q_1^k atomic. Replace the original equation by the formula

$$\Big(\bigwedge_a \sum_{p_1^i = a} p_2^i = \sum_{q_1^k = a} q_2^k\Big) \wedge \Big(\bigwedge_j \bigvee_l p_3^j = q_3^l\Big) \wedge \Big(\bigwedge_l \bigvee_j p_3^j = q_3^l\Big) \qquad (7)$$

(where a conjunction means that all conjuncts are valid, and a disjunction that one of the disjuncts is valid (!)).

By construction, the depth of the deepest occurrence of ret in the target equations has been decreased by one in comparison to the original equation, if originally it was greater than one. If it was equal to one, because of (2) the whole routine might have to be repeated once in order to decrease it to zero.

Soundness of steps (1) and (2) is clear, because they only appeal to the rules of MCE. Let us now show that step (3) is also legitimate. Indeed, by Lemma 10, for any a, $\delta_{x\leftarrow a}(p) = \delta_{x\leftarrow a}(q)$. The easily verifiable identity $t = \sup \delta_{x\leftarrow a}(\mathrm{do}\ x \leftarrow a; t)$ shows that the first conjunct in (7) holds, because it represents equalities of suprema of equal sets. Then note that $\varepsilon(p)$ and $\varepsilon(q)$ are finite. By Lemma 10, they must be elementwise equal, which is expressed by the second and the third conjunct of (7).

Now suppose that p and q are both ret-free. W.l.o.g. assume that all the variables in p and q are bound (free ones can be seen as constants). Assign to every normal atomic program a a new functional symbol $\hat{a}$. Let $\widehat{\Sigma}$ be the set of all such symbols. The latter can be seen as an extension of Σ, because any functional symbol from Σ is a special case of an atomic program. Define a map ι, taking every ret-free program to a term of Kleene algebra over $\widehat{\Sigma}$, as follows:

- $\iota(a) = \hat{a}$, for atomic a,
- $\iota(\emptyset) = \emptyset$,
- $\iota(p + q) = \iota(p) + \iota(q)$,
- $\iota(\mathrm{do}\ x \leftarrow p; q) = \iota(q) \cdot \iota(p)$,
- $\iota(\mathrm{init}\, x \leftarrow p \,\mathrm{in}\, q^*) = \iota(q)^* \cdot \iota(p)$.

By differentiating the equation $p = q$ appropriately many times, one can show that for any atomic $a_1, \ldots, a_n$, if $\mathrm{do}\ a_1; \ldots; a_n \leq p$, then $\mathrm{do}\ a_1; \ldots; a_n \leq q$ and vice versa. This means precisely that $\iota(p) = \iota(q)$ holds over the algebra of regular events. Hence, by the completeness result of [9], the last equation must be provable in the corresponding calculus for Kleene monads. We are done if we show how to turn this proof into a proof in MCE. To this end, define an operator κ_X taking any Kleene algebra term to a term of the MCE of type $X \to TX$ over the signature $\widehat{\Sigma}$ by

- $\kappa_X(\hat{a}) = \hat{a}(z)$, $\hat{a} \in \widehat{\Sigma}$
- $\kappa_X(\emptyset) = \emptyset$,
- $\kappa_X(p + q) = \kappa_X(p) + \kappa_X(q)$,
- $\kappa_X(p \cdot q) = \mathrm{do}\ z \leftarrow \kappa_X(q); \kappa_X(p)$,

– $\kappa_X(p^*) = \mathrm{ret}\, z + \mathrm{init}\, z \leftarrow \kappa_X(p) \,\mathrm{in}\, \kappa_X(p)^*$.

If we apply κ_X to the proof at hand, we obtain a valid proof in MCE of the equality $\kappa_X(\iota(p)) = \kappa_X(\iota(q))$ – this follows from the obvious similarity between MCE and the calculus for Kleene algebras. The idea is to pick out an interpretation of a symbol from $\widehat{\Sigma}$ in such a way that the last equation turns into $p = q$. W.l.o.g. assume that every variable of p and q is bound only once. Let $\Delta = (x_1 : TA_1, \ldots, x_n : TA_n)$ be the context of all variables occurring in p and q. Put $X = TA_1 \times \ldots \times TA_n \times TA$, where TA is the common type of p and q. Recall that every $\hat{a} \in \widehat{\Sigma}$ occurring in $\iota(p)$, $\iota(q)$ corresponds to an atomic program $(x_{k_1} : TA_{k_1}, \ldots, x_{k_m} : TA_{k_m}) \rhd a : TB$ occurring in p, q, whose return value is either bound to some variable x_k or propagated to the top of the term. In the latter case, we just assume $k = n + 1$. Now put

$$\hat{a}(z) = \mathrm{do}\ x \leftarrow a\langle \pi_{k_1} z, \ldots, \pi_{k_n} z\rangle; \mathrm{ret}\langle \pi_1 z, \ldots, \pi_{k-1} z, x, \pi_{k+1}, \ldots, \pi_{n+1} z\rangle.$$

Having defined the interpretation of all the symbols $\hat{a}$ in this manner, we obtain an equation over the original signature Σ. It can be shown by induction that the original equation $p = q$ can by obtained from it by application of the operator $\lambda t.\, \mathrm{do}\ z \leftarrow t; \mathrm{ret}\, \pi_{n+1} z$. Decidability follows from the algorithmic character of the reduction steps (1) – (3) and the fact that Kleene algebra is decidable. □

The definition of regularity might seem overly restrictive. It is easy to find examples of programs that do not satisfy regularity directly, but are still equivalent to a regular program. A typical example of this kind is $\mathrm{init}\, x \leftarrow p \,\mathrm{in} (\mathrm{do}\ x \leftarrow q; \mathrm{ret}\, a)^*$, which is equivalent to $p + \mathrm{do}\ x \leftarrow (\mathrm{init}\, x \leftarrow q \,\mathrm{in}\, q[a/x]^*); \mathrm{ret}\, a$. But in fact, it is not even semi-decidable to check whether a program admits a regular implementation.

Theorem 15. *The problem of checking whether a program in the MCE is semantically equivalent to a regular program over ω-continuous Kleene monads is non-r.e.*

5 Worked Example: Stack Reverse in the MCE

In spite of the negative result proved in Theorem 12 we believe the calculus MCE to be a reasonable framework for proving program equivalence. In fact, we have encoded the calculus in the Isabelle/HOL prover. We now present an example that we have axiomatised as an Isabelle theory and successfully verified[3]: the double-reverse theorem stating that reversing a list twice yields the original list. More specifically, we have proved this theorem for two implementations of reverse, the single-stack example from the introduction and a further implementation using two stacks. For the latter, we declare operations

$$\mathrm{pop}_i : TA, \qquad \mathrm{push}_i : A \to T1, \qquad \mathrm{is_empty}_i : T1.$$

[3] The theory files are available under `http://www.informatik.uni-bremen.de/~sergey/StackReverse.tar`

($i \in \{1,2\}$). Some of the axioms imposed on these are

$$\begin{array}{rr} \mathrm{do}\ \mathrm{is_empty}_i; \mathrm{pop}_i = \emptyset & \mathrm{do}\ x \leftarrow \mathrm{pop}_i; \mathrm{push}_i\, x \le \mathrm{ret}\langle\rangle \\ \mathrm{do}\ \mathrm{push}_i\, x; \mathrm{is_empty}_i = \emptyset & \mathrm{do}\ \mathrm{push}_i\, x; \mathrm{pop}_i = \mathrm{ret}\, x \\ \mathrm{do}\ \mathrm{is_empty}_i; \mathrm{is_empty}_i = \mathrm{is_empty}_i & \mathrm{is_empty}_i \le \mathrm{ret}\langle\rangle \end{array}$$

Define $\mathrm{append}_{ij} = \mathrm{do}\ \mathrm{init}\,\mathrm{ret}\langle\rangle\,\mathrm{in}(\mathrm{do}\ x \leftarrow \mathrm{pop}_i; \mathrm{push}_j\, x)^*; \mathrm{is_empty}_i$, which pops elements from the i-th stack and pushes them into j-th stack in reverse order until the i-th stack is empty. Now the double reverse theorem for two stacks can be encoded as

$$\mathrm{do}\ \mathrm{is_empty}_2; \mathrm{append}_{12}; \mathrm{append}_{21} \le \mathrm{is_empty}_2$$

(where the inequality arises because there is no axiom that guarantees that the stack bottom is reachable, and expresses primarily that the left hand program does not change the state). The proofs are reasonably straightforward but require some degree of manual intervention; the possibility of better proof automation is under investigation.

6 Conclusion and Future Work

We have combined Moggi's computational metalanguage for monads and Kozen's axiomatisation of Kleene algebra to obtain a metalanguage of control and effects (MCE) that can be interpreted over a suitable notion of Kleene monad. While the combined equational language cannot be recursively axiomatised, a quite expressive sublanguage of so-called regular programs does have a complete axiomatisation. This axiomatisation has been formalised in the theorem prover Isabelle/HOL, and some sample verifications have been performed on programs involving monadic do-notation, Kleene star, and nondeterminism, as well as the use of programs as values in a higher-order style. The MCE forms part of an evolving system of formalisms (see e.g. [17, 16, 14]) aimed at the development of a verification framework for functional-imperative programs in the style of Haskell. The further development of this framework will include the integration of Kleene monads with monad-based dynamic logic [17, 14]; an extension of the MCE with side effect free tests in analogy to Kleene algebra with tests [10], which will in particular accommodate the Fischer-Ladner encoding of `if` and `while`; and parametrisation of the MCE over an underlying equational theory of data. A further open point is how our approach to the combination of iteration and effects can be transferred to the related framework of algebraic effects, in particular w.r.t. the combination of effects [5].

Acknowledgement We thank Erwin R. Catesbeiana for finite discussions about infinite iterations.

References

[1] P. N. Benton, G. M. Bierman, and V. de Paiva. Computational types from a logical perspective. *J. Funct. Prog.*, 8(2):177–193, 1998.

[2] P.-L. Curien and R. di Cosmo. A confluent reduction for the lambda-calculus with surjective pairing and terminal object. *J. Funct. Program.*, 6(2):299–327, 1996.

[3] M. Fluet, G. Morrisett, and A. J. Ahmed. Linear regions are all you need. In *Programming Languages and Systems, ESOP 2006*, vol. 3924 of *LNCS*, pp. 7–21. Springer, 2006.

[4] R. Hinze. Deriving backtracking monad transformers. *ACM SIGPLAN Notices*, 35(9):186–197, 2000.

[5] M. Hyland, G. D. Plotkin, and J. Power. Combining effects: Sum and tensor. *Theoret. Comput. Sci.*, 357:70–99, 2006.

[6] B. Jacobs and E. Poll. Coalgebras and Monads in the Semantics of Java. *Theoret. Comput. Sci.*, 291:329–349, 2003.

[7] J.-P. Jouannaud and M. Munoz. Termination of a set of rules modulo a set of equations. In *Automated Deduction, CADE 1984*, vol. 170 of *LNCS*, pp. 175–193. Springer, 1984.

[8] O. Kiselyov, C. Shan, D. Friedman, and A. Sabry. Backtracking, interleaving, and terminating monad transformers. In *Functional Programming, ICFP 2005*, pp. 192–203. ACM, 2005.

[9] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Inf. Comput.*, 110:366–390, 1994.

[10] D. Kozen. Kleene algebra with tests and commutativity conditions. In *Tools and Algorithms for Construction and Analysis of Systems, TACAS 1996*, LNCS, pp. 14–33. Springer, 1996.

[11] D. Kozen. Nonlocal flow of control and Kleene algebra with tests. In *Logic in Computer Science, LICS 2008*, pp. 105–117. IEEE, 2008.

[12] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93:55–92, 1991.

[13] E. Moggi and A. Sabry. Monadic encapsulation of effects: A revised approach (extended version). *J. Funct. Prog.*, 11:591–627, 2001.

[14] T. Mossakowski, L. Schröder, and S. Goncharov. A generic complete dynamic logic for reasoning about purity and effects. In J. Fiadeiro and P. Inverardi, eds., *Fundamental Approaches to Software Engineering (FASE 2008)*, vol. 4961 of *Lecture Notes in Computer Science*, pp. 199–214. Springer, 2008. Extended version to appear in *Formal Aspects of Computing*.

[15] S. Peyton Jones, ed. *Haskell 98 Language and Libraries — The Revised Report*. Cambridge University Press, 2003. Also: J. Funct. Prog. **13** (2003).

[16] L. Schröder and T. Mossakowski. Generic exception handling and the Java monad. In *Algebraic Methodology and Software Technology, AMAST 2004*, vol. 3116 of *LNCS*, pp. 443–459. Springer, 2004.

[17] L. Schröder and T. Mossakowski. Monad-independent dynamic logic in HASCASL. *J. Logic Comput.*, 14:571–619, 2004.

[18] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time: Preliminary report. In *Symposium on Theory of Computing, STOC 1973*, pp. 1–9. ACM, 1973.