Kleene Monads:
Handling Iteration in a Framework of Generic Effects

Sergey Goncharov1, Lutz Schröder1,2, and Till Mossakowski1,2,*

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 non-deterministic choice and Kleene star, i.e. non-deterministic 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 is_empty, push, and pop be the usual stack operations, where is_empty blocks if the state is nonempty, and otherwise does nothing. Then one can define the reverse operation in the MCE as

\[
\text{do } q \leftarrow (\text{init } p \leftarrow \text{ret is_empty in (do } x \leftarrow \text{pop; ret (do } p; \text{push } x))^{*}) ; q
\]

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

\[
\text{do } q \leftarrow (\text{do } x_1 \leftarrow \text{pop}; \ldots ; x_n \leftarrow \text{pop; ret(do is_empty; push } x_1; \ldots ; \text{push } x_n)) ; q
\]