Formalising Algebraic Effects with Non-Recoverable Failure

Timotej Tomandl
University of Kent

Dominic Orchard
University of Kent

Abstract

Algebraic effects are one approach to tackling modularity issues arising in structuring side effects. One such application area is in (formalised) programming language semantics. We observe that in this setting, there is a need to separate the model of effects of the language being formalised from the effects arising as incidental artefacts of the formalisation. In this work, we focus on failure as such an incidental effect. We thus seek an extension to the usual underlying mathematical structures for capturing algebraic effects to incorporate (non-handleable) failure. We show how the MaybeT monad transformer over a free monad construction can be used, studying the implications on the underlying theory via a (work in progress) mechanisation in type theory. Our aim is to host both the language formalisation and its associated theory within one mechanised framework. Along the way, we also give a graded monadic view of algebraic effects.

1 Motivation

Consider the situation of a writing a verified compiler with a reference interpreter where initial typing information has been erased. This necessitates some error handling cases which are known to not occur (assuming that the typing procedure for this language is sound):

```
... UnOpNegate e -> case eval env e of
                      Num n -> Num (- n)
                      _     -> error "Impossible"
```

The `error` expression here represents a failure side-effect that is non-recoverable; once it is evaluated, failure prevents the rest of the interpretation.

In Haskell, the above code pattern can be abstracted via the *do*-notation and the use of the `MonadFail` type class which provides an operation `fail :: String -> m a` which is used to desugar pattern matches which may fail. In this style, the above code could be written as:

```
... UnOpNegate e -> do
        Num n <- eval env e
        return (Num (- n))
```

Since the `(Num n)` pattern is 'refutable' this desugars into:

```
  eval env e >>= (\v -> case v of
      Num n -> return (Num (- n))
      _     -> fail "Incomplete pattern match")
```

This failure acts as a kind of *exception*. Exceptions, and their handling, are generalised by notions of *algebraic effects* and *handlers* which can be used to give a modular semantics to side effects. The language being modelled may also have other notions of side effect, like state, which could then be captured via an algebraic effect approach. However, the above notion of failure is not a side effect of the language but rather an incidental feature of the formalisation. Therefore, it should not be conflated with the model of exception side effects in the language being modelled/formalised.

In this work, we capture the idea of `MonadFail` in the algebraic effects approach, but where failure is an intrinsic non-handleable effectful operation, seen as orthogonal to the usual effect operations.

Failure of this form has been a staple feature of *monad transformer frameworks* [Liang et al. 1995], which can express the above pattern. Monad transformers do not provide as much flexibility as algebraic effects [Kiselyov and Ishii 2015] and need more complex code to express many patterns. We explore how to combine these two ideas.

Our development is formalised in dependent-type theory (we use Agda [Tomandl 2021]) capturing mathematical results of the key constructions. As part of this, we contribute our own type theoretic rendering of free monads (based on a variation of Kiselyov and Ishii [2015]'s 'freer' monads) which is indexed by the effect operations captured within the computation tree describe by a free monad value. Our approach yields a graded monad, which provides a way to reason about the compositional nature of effects with failure.

2 Underlying model for algebraic effects

Algebraic effects in theory consist of a signature of operations and their equations. The collection of operations can be represented type theoretically by a parametric data type F combined with the free monad construction ($\mu X.A + FX$) to capture computations trees of effect operations which can be 'handled' later. However, this free monad definition is not amenable to formalisation in type theory, because its type-level fixed point can lead to inconsistency. Most tools disallow this free monad definition via positivity checks.

However, the free endofunctor generated by the left Kan extension on any type constructor avoids this positivity issue. This is captured by Kiselyov and Ishii [2015]'s 'freer' monad construction, defined (in Haskell) as:

```
data FFree (f :: Type -> Type) (a :: Type) where
  Pure   :: a -> FFree f a
  Impure :: f x -> (x -> FFree f a) -> FFree f a
```

[0]Extended abstracted presented at HOPE'21

The `Impure` constructor captures effectful operations described by the type constructor `f`. Thus these effect operations captured by `f` must be determined globally for the computation.

We use a more compositional generalisation, allowing subcomputations to use different type constructors to capture their effect operations. A type index gathers these various type constructors as a list. This is an intermediate design between the constructions `FFree` and `Eff` in Kiselyov and Ishii [2015]. In Agda, we define this as:

```
data Eff (E : List (Set -> Set)) (A : Set) : Set₁ where
  Pure   : A -> Eff E A
  Impure : {F : Set -> Set} {X : Set} {prf : F ∈ E}
        -> F X -> (X -> Eff E A) -> Eff E A
```

where `prf` witnesses that `F` is in the indexing list `E`.

Proposition 2.1. *For all* `E`*,* `Eff E` *is a monad.*

We expose the compositionality of this approach via a *graded monad* construction. Graded monads comprise an indexed family of endofunctors $\{M_x\}_{x \in X}$, whose indices are the elements of a monoid $(X, I, \bullet)$, with unit $\eta : \mathsf{Id} \longrightarrow MI$ and multiplication $\mu : Mx \circ My \longrightarrow M(x \bullet y)$ operations akin to monad operations (with analogous equations to a monad) [Katsumata 2014; Orchard et al. 2014]. Thus the monoid explains that η captures pure computations and μ sequentially composes computations, combining their effects.

Proposition 2.2. `Eff` *is a graded monad indexed by the list monoid with* $\eta : \mathsf{Id} \longrightarrow M[]$ *and* $\mu : Me \circ Mf \longrightarrow M(e +\!\!+ f)$*.*

The graded monad construction suggests combinators for raising and handling effects where grades give an account of what needs handling (where `E - F` deletes `F` from the list `E`):

```
raise  : F a -> Eff [F] a
handle : (∀ x -> F x -> x) -> Eff E a -> Eff (E - F) a
```

We can also 'run' a computation once all effects are handled:

```
run : Eff [] a -> a
```

3 MonadFail and exceptions

In Haskell, if a *do*-expression performs a pattern matching 'bind' that may fail, then the monad being used must also have an instance of the `MonadFail` class:

```
class MonadFail m where
  fail :: String -> m a
```

where, for all monadic computations `x :: m a`:

$$\texttt{fail} >\!\!>\!\!= \texttt{x} \;\equiv\; \texttt{fail}$$

i.e., failure is global and non-recoverable.

We give an abstract mathematical definition, ignoring the `String` aspect in favour of a map from the terminal object 1.

Definition 3.1 (Failing monad)**.** For a category C with terminal object 1, a *failing monad* for endofunctor $T : C \longrightarrow C$ is a monad with a natural transformation $fail_A : 1 \longrightarrow T\ A$ such that for all $g : 1 \longrightarrow T\ B$ then:

$$\mu \circ T(g \circ !_A) \circ fail_A = fail_B$$

Exceptions are the prototypical example of an effect handler [Plotkin and Pretnar 2009]. We can instantiate `Eff E` to yield a failing monad with the following effect operation (where `Void` is the empty type):

```
data Error : Set -> Set where Raise : Error Void

fail : {A : Set} {E : List (Set -> Set)}
     .{prf : Error ∈ E} -> Eff E A
fail {A} {E} {prf} =
      Impure {F = Error} {X = Void} {prf} Raise (\())
```

Proposition 3.2. `Eff E` *where* `Error in E` *with the above definition of* `fail` *is a failing monad.*

However, we want to make failure an intrinsic part of underlying appartus of algebraic effects, that is part of the free monad, such that it is not handleable. If we attempt to give a `MonadFail` instance for the free monad, we get stuck immediately: the free monad, is the monad for which exactly the monad equations hold and nothing more, i.e., not the extra `MonadFail` axiom above. Therefore, we seek to have both the benefits of the free monad structure for capturing computation trees (for the purposes of effect handling) but also the property of non-recoverable failure captured by `MonadFail`.

4 MaybeT for non-handleable failure

The simplest construction to support the failing monad law generally is provided by the "`MaybeT`" *monad transformer* [Liang et al. 1995], in Haskell defined as:

```
newtype MaybeT m a = MaybeT (m (Maybe a))
```

That is, a `MaybeT` is a monad homomorphism, mapping $M-$ to $M(1 + -)$. We can think of `MaybeT` as an extension by an arbitrary, single option failure case. The `MaybeT` transformer gives a lawful instance of `MonadFail` for all monads:

```
  instance Monad m => MonadFail (MaybeT m) where
    fail _ = MaybeT (return Nothing)
```

We can thus apply this idea to our `Eff` construction, yielding a failing monad `MaybeT (Eff F)`. We call this construction `EffectFail F`. However, we still require *freeness* of `EffectFail` in order to yield a suitable semantic core for handling algebraic effects (augmented with non-recoverable failure). `EffectFail`. We thus ask, *is* `EffectFail` *the free failing monad?*.

To accomodate the failure case, we have to modify the notion of handling. In standard form initiality gives us a notion of interpreting a monad `Free f` by giving a way to iterate over `f`. In other words given a function of type `f x -> x` we get a function `Free f x -> x`. Thus, we state a different form of initiality. Initiality can be thought of as a principle, $(F\ A \longrightarrow A) \longrightarrow (\mathsf{Free}\ F\ A \longrightarrow A)$, which allows us to write a fold over the abstract syntax of monads given a fold over the underlying datatype. However, the correct handling for free `EffectFail` is $(F\ A \longrightarrow A) \longrightarrow$ `EffectFail` $F\ A \longrightarrow$ `Maybe` A. As much as effects are inspired by initiality, we provide a type-theoretic motivation for our definition. To provide handling of `EffectFail`, we interpret the results and embed them into `Maybe` and propagate the underlying failure. Thus this is initiality not for `EffectFail`, but for `Eff` itself after expanding the `MaybeT` monad transformer. This should be correct not by some appeal to category theory, as initiality is normally stated, but by appealing to the API we would expect from our system. To run `EffectFail` we either want to extract the underlying result or propagate the failure of an underlying `Pure` result.

This construction shouldn't be too much of a surprise since `MaybeT` is known to provide the underlying monad for an algebraic theory adding exceptions [Hyland et al. 2006]. What is new here is that we consider this failing monad as the underlying apparatus for algebraic effects, rather than arising for a particular kind of algebraic effect theory.

Algebraic freeness. Being able to interpret effects is not the only property of interest for free monads, we also want to characterise the algebras of the free monad with respect to the underlying functor. Where initiality, also called freeness, gives us a way to handle effects, its namesake algebraic freeness gives us a way to obtain this characterisation.

A monad T is *algebraically free on an endofunctor F* if the category of T-algebras (monad algebras) is isomorphic to the category of F-algebras. An algebraically-free monad is itself a free monad [Kelly 1980; nLab authors 2021]. We conjecture that some (perhaps extended) notion of algebraic freeness can also be established for `EffectFail F` showing that it is the "free" structure here.

5 Future directions

We have constructed the `Eff` monad in Agda and verified the various monad / graded monad laws. For practical purposes of reasoning there needs to be more than just a monadic structure: we want to reason about the code using the eponymous algebraic laws. Therefore, we would like to attach to the monad the equations that the handlers should satisfy and thus provide correct-by-construction handlers. One solution for attaching equations to a datatype are Quotient Inductive Types [Altenkirch and Kaposi 2016]. This should abstract away from the concrete choices when reasoning, yet force us to prove satisfiability of the algebraic equations by some handler. To extend the proofs from our current work to the quotient case, we need to prove the type-theoretic formulation of algebraic freeness (discussed above) in order to prove that the free monad constructions preserve all equations. Algebraically-free monads are free, and all free monads are algbraically-free in locally small and complete categories [nLab authors 2021]. This result should hold in (homotopy) type theory as we should have all colimits and right Kan extensions, which are used in such proofs as consequences of the locally small and complete structure.

These lemmas apply only to the free monad with respect to a forgetful functor $Mnd(C) \to Endo(C)$. We suggest a generalisation to the forgetful functor $Mnd(C) \to [ob(C), C]$ where $[ob(C), C]$ is the category of functors mapping from the discrete category of C to C, matching more closely the left Kan construction used by `Eff`. However, there is no obvious generalisation, which would let us talk about algebras over an arbitrary $[ob(C), C]$. We thus take a more type theoretic view to extend the isomorphism in the definition of algebraic freeness to the freer construction.

The semantics of a type constructor F is thought as an initial object in some category $Alg(F)$ corresponding to its algebraic description. Homotopy type theory generalizes this to quotient types (and other more general higher inductive types), thus we should be able to present F-algebras, including their equations, using quotient types. Informally the left Kan extension, $Lan(F)$, of a datatype F should give us an endofunctor, which just accumulates functions applied to it without actually applying them, postponing their interpretation. We propose that the algebras of F are isomorphic to the algebras of the left Kan extension of F:

Conjecture 5.1 (Algebras of Left Kan extensions)**.** *Given a type family $F : Set \to Set$ with a homotopy-initial algebra semantics $Alg(F)$, the family of $Lan(F)$-endofunctor algebras is isomorphic to $Alg(F)$.*

If this conjecture holds, then the 'freer' monad T over X would have T-algebras isomorphic to the initial algebras of X. This follows by chaining this conjecture with the statement of algebraic freeness. We expect the formal statement of this conjecture to be provable with some appropriate container [Altenkirch et al. 2015] or generic levitated presentation [Chapman et al. 2010], because this would allow us to connect the $\mathsf{Lan}(F)$-algebra of an endofunctor with the algebra of the underlying datatype F.

Another direction of future research is to construct an isomorphism between continuations and the fast type-aligned queues of Kiselyov and Ishii [2015]; Ploeg and Kiselyov [2014], quotiented by their composition, thus providing a more efficient implementation of effects without the overheads to reasoning this entails as we focus on semantic constructions instead of executable code, this is not our primary concern. We would expect this to then transport by univalence, but as discussed in [Tabareau et al. 2018] we would need some form of representation independence coming from parametricity, for this to be actually efficient.

Lastly, we have yet to connect the graded monad story with failing monads. There is a straightforward extensions of `MaybeT` to a graded setting, e.g., transforming the grading monoid X to a product monoid $X \times \mathbb{B}$ where $\mathbb{B}$ represents whether a computation is defined or possibly undefined. Exploring this and richer alternatives is ongoing work.

References

T. Altenkirch and A. Kaposi. Type theory in type theory using quotient inductive types. *ACM SIGPLAN Notices*, 51(1):18–29, 2016.

T. Altenkirch, N. Ghani, P. Hancock, C. McBride, and P. Morris. Indexed containers. *Journal of Functional Programming*, 25, 2015.

J. Chapman, P. Dagand, C. McBride, and P. Morris. The gentle art of levitation. In P. Hudak and S. Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 3–14. ACM, 2010. doi: 10.1145/1863543.1863547.

M. Hyland, G. Plotkin, and J. Power. Combining effects: Sum and tensor. *Theoretical computer science*, 357(1-3):70–99, 2006.

S. Katsumata. Parametric effect monads and semantics of effect systems. In S. Jagannathan and P. Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 633–646. ACM, 2014. doi: 10.1145/2535838.2535846. URL https://doi.org/10.1145/2535838.2535846.

G. Kelly. A unified treatment of transfinite constructions for free algebras, free monoids, colimits, associated sheaves, and so on. *Bulletin of the Australian Mathematical Society*, 22(1):1–83, 1980. doi: 10.1017/S0004972700006353.

O. Kiselyov and H. Ishii. Freer monads, More Extensible Effects. *SIGPLAN Not.*, 50(12):94–105, Aug. 2015. ISSN 0362-1340. doi: 10.1145/2887747.2804319. URL https://doi.org/10.1145/2887747.2804319.

S. Liang, P. Hudak, and M. P. Jones. Monad transformers and modular interpreters. In R. K. Cytron and P. Lee, editors, *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 333–343. ACM Press, 1995. doi: 10.1145/199448.199528.

nLab authors. free monad. http://ncatlab.org/nlab/show/free%20monad, May 2021. Revision 15.

D. A. Orchard, T. Petricek, and A. Mycroft. The semantic marriage of monads and effects. *CoRR*, abs/1401.5391, 2014.

A. v. d. Ploeg and O. Kiselyov. Reflection without remorse: revealing a hidden sequence to speed up monadic reflection. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, pages 133–144, 2014.

G. Plotkin and M. Pretnar. Handlers of algebraic effects. In *European Symposium on Programming*, pages 80–94. Springer, 2009.

N. Tabareau, E. Tanter, and M. Sozeau. Equivalences for free: Univalent parametricity for effective transport. *Proc. ACM Program. Lang.*, 2(ICFP), July 2018. doi: 10.1145/3236787. URL https://doi.org/10.1145/3236787.

T. Tomandl. effect-fail: Agda implementation. https://github.com/formrre/effect-fail/tree/HOPE, 2021.