

Modular Monadic Reasoning, a (Co-)Routine

Steven Keuchel and Tom Schrijvers

Universiteit Gent, Belgium,
`{steven.keuchel,tom.schrijvers}@ugent.be`

Abstract. Higher-order functions that are polymorphic in a monad make highly flexible modular components. Unfortunately, the combination of an unknown function parameter and a polymorphic monad are detrimental to reasoning. This paper shows how to eliminate both the function parameter and the polymorphism. The resulting characterization is amenable to reasoning.

The approach is based on a judicious combination of the coroutine monad transformer and monad morphisms.

1 Introduction

Modularity is one of the holy grails of software engineering. The dream is to be able build new software systems entirely from reusable components, that have been written independently and can (potentially) be reused in many different configurations for different applications. Increasingly more modularity demands are being made: without modifying components, it must be possible to augment or modify their behavior. At the same time, unprincipled copy-&-paste approaches, even performed by automated tools, are not acceptable. Components must have a meaning beyond their textual form and independently of a particular composition; they must support *modular reasoning*.

This paper considers modular reasoning in the purely functional setting of polymorphic monadic mixin components. Mixin components [1] employ a form of dynamic inheritance, called open recursion, to make a function's behavior modifiable. Monads [2] enable components to have side effects. Polymorphic monads do not fix the side effects up front, but enable different components in a composition to contribute their own side effects to the whole.

These polymorphic monadic mixin components are extremely flexible in their use. Yet, they prove to be highly challenging with respect to modular reasoning. Oliveira et al. [3] show how to do reasoning about non-interference of such components based on parametricity, i.e., based on the types of the components alone. However, modular reasoning about more involved, implementation-dependent properties is still an open problem.

Modular reasoning is difficult in this setting, because it combines two programming features that are independently difficult to reason about, but absolutely fiendish together: higher-order functions and polymorphic monads. Independently, they can be tackled with equational reasoning, free theorems and

monad laws, but these tools have only limited traction on higher-order functions over polymorphic monads.

Inspired by Hofmann et al.’s work [4,5] on characterizing pure monadic higher-order functions, this paper shows how to eliminate the polymorphism and the higher-order parameter for a range of side effects. Hence, we characterize polymorphic monadic mixin components with monomorphically-typed first-order representations. We believe that these representations are inherently more convenient for reasoning.

As this is work in progress, and deriving the first order representation is challenging in its own right, modular reasoning is not covered.

2 Motivating Example

This is the core infrastructure for mixin components:

```
type Open s = s → s
new :: Open s → s
new a = a (new a)
(⊕) :: Open s → Open s → Open s
a1 ⊕ a2 = a1 ∘ a2
```

A mixin component of type `Open s` has a hole of type `s` for the recursive occurrences. This hole is closed with fixpoint combinator `new`. Two mixin components are composed with `⊕`.

Here is an example, the Fibonacci function rendered as a polymorphic monadic mixin component:

```
fib :: Monad m ⇒ Open (Int → m Int)
fib rec n | n < 2 = return n
fib rec n | otherwise = do x ← rec (n - 1)
                          y ← rec (n - 2)
                          return (x + y)
```

As `m` can be any monad, `fib` does not rely on any side effect. We say that `fib` is a *pure component*. The (slow) Fibonacci function is recovered by closing the open component and instantiating `m` to the identity monad:

```
slowfib n = runId (new fib n)
```

Note that this paper uses the Monatron library [6] for monads and monad transformers. Monatron provides superior lifting capabilities compared to other monad transformer libraries that we will use later in this paper.

A faster implementation of the Fibonacci function is obtained with the help of memoization. The memoization functionality can be captured orthogonally in its own component:

```

memo :: (Eq k, StateM [(k, v)] m) => Open (k -> m v)
memo super n =
  do t <- get
  case lookup n t of
    Just v  -> return v
    Nothing -> do v <- super n
                modify ((n, v):)
                return v

```

This component uses the state side effect, which is documented in the constraint `StateM [(k, v)] m`. We obtain an $\mathcal{O}(n^2)$ -time Fibonacci function by composing `memo` with `fib`, closing the open recursion and running the resulting function in the `State` monad.

```
fastfib n = evalState [] (new (memo ⊕ fib) n)
```

2.1 Modular Reasoning

To show the validity of applying memoization, the following property must be established:

$$\text{slowfib} \equiv \text{fastfib}$$

Oliveira et al. [7] prove this property in a non-modular fashion: they consider the composition `memo ⊕ fib` as a whole. Our goal is to reason about this composition modularly: We want to 1) establish properties about `memo` and `fib` based on their respective implementations but independent of one another, and 2) combine these properties to establish the above property without further reliance on the implementations.

The individual properties should be reusable for establishing a range of properties:

1. The `memo` mixin is non-interfering for any pure component $f :: \text{Monad } m \Rightarrow \text{Open } (i \rightarrow m o)$:

$$\text{evalState [] (new (memo} \oplus f) n) \equiv \text{runld (new f n)}$$

2. The application of other mixins to `fib`, that implement alternative memoization techniques, is correct.
3. Adding particular additional effects, e.g., logging, does not affect the correctness of memoization:

$$\text{fst \$ evalState [] \$ runWriterT (new (log} \oplus \text{memo} \oplus \text{fib) n) \equiv \text{runld (new fib n)}$$

This is very challenging, because different uses require different instantiations of the monad type parameter `m`, and the implementation-dependent property must cater to all at once. This rules out expanding the definitions of the monadic

operations for any particular instance. A second complicating factor is the function parameter for recursive or super calls. Its particular behavior (side-effects and returned value) depend on the composition. In the non-monadic setting, we can formulate preconditions on the input-output behavior of the parameter function. However, in this monadic setting, and due to the polymorphism, the function parameter may have access to more or other side-effects than the mixin itself. It is not clear how to impose any preconditions on these unknown side-effects or how to factor them into the overall behavior of the mixin.

3 Strategy Trees for Pure Functionals

The starting point of this paper are the results of Bauer et al. [5], who have studied the characterization of pure functionals of type `Func i o a`:

```
type Func i o a =  $\forall m. \text{Monad } m \Rightarrow (i \rightarrow m\ o) \rightarrow m\ a$ 
```

They show that such functionals “can be seen as strategies in a question-answer game leading to the computation of” the result. These strategies are reified in *strategy trees*:

```
data Tree i o a where
  Ans :: a  $\rightarrow$  Tree i o a
  Que :: i  $\rightarrow$  (o  $\rightarrow$  Tree i o a)  $\rightarrow$  Tree i o a
```

A strategy tree `Tree i o a` for a pure functional of type `Func i o a` is a sequence of queries `Que` resulting in an answer `Ans`. Bauer et al. show that there exists a strategy tree for every functional, and that it is constructed by `fun2tree`:

```
fun2tree :: Func i o a  $\rightarrow$  Tree i o a
fun2tree f = runCont Ans (f ( $\lambda i \rightarrow$  cont ( $\lambda k \rightarrow$  Que i k)))
```

Moreover, `tree2fun` recovers the functional.

```
tree2fun :: Tree i o a  $\rightarrow$  Func i o a
tree2fun t f = go t where
  go (Ans a) = return a
  go (Que i k) = f i  $\gg$  go  $\circ$  k
```

Note that the `fun2tree` and `tree2fun` functions are each other’s inverses:

```
tree2fun  $\circ$  fun2tree  $\equiv$  id
fun2tree  $\circ$  tree2fun  $\equiv$  id
```

3.1 Strategy Trees for Monadic Mixins

It turns out that strategy trees are not restricted to functionals of type `Func i o a`, but are also suitable for encoding monadic mixins. For instance, `fibt` defines strategy tree of `fib`.

```
fibt :: Int → Tree Int Int Int
fibt n | n < 2 = Ans n
fibt n       = Que (n - 1) (λf1 →
                        Que (n - 2) (λf2 →
                        Ans (f1 + f2)))
```

More formally, we can state the following equivalence:

$$\text{fib} \equiv \text{tree2open fibt}$$

where `tree2open` is a variant of `tree2fun` that is more convenient for open components:

```
tree2open :: Monad m ⇒ (i → Tree i o o) → Open (i → m o)
tree2open f super i = tree2fun (f i) super
```

More generally, any pure monadic mixin component of type $\forall m. \text{Monad } m \Rightarrow \text{Open } (i \rightarrow m \ o)$ can be encoded by a strategy tree of type `Tree i o o`:

```
open2tree :: (∀m. Monad m ⇒ Open (i → m o)) → (i → Tree i o o)
open2tree h i = fun2tree (flip h i)
```

This is great for reasoning, because the latter type is monomorphic, while the former is not.

Unfortunately, this observation far from solves our problems as it does not cover monadic mixins like `memo`. Unlike `fib`, `memo` is not a pure monadic component. On the contrary, it explicitly relies on state to implement its behavior. Thus it cannot be encoded as a strategy tree, and we are no step further to modularly reasoning about components with side effects.

4 Stateful Strategy Trees

In order to make progress on reasoning about `memo`, this section explores a variant of Bauer's pure strategy trees that caters for functionals of type:

```
type FuncS s i o a = ∀m. StateM s m ⇒ (i → m o) → m a
```

This stateful strategy tree type is:

```
data TreeS s i o a where
  AnsS :: a → s → TreeS s i o a
  QueS :: i → s → (o → s → TreeS s i o a) → TreeS s i o a
```

A strategy is either an answer Ans_S or a query Que_S . The constructor Ans_S captures the final state alongside the answer. Similarly, Que_S exposes the intermediate state and its continuation takes an updated state. The stateful strategy tree for `memo` is:

```
memot :: Eq k => k -> [(k, v)] -> Tree_S [(k, v)] k v v
memot n t = case lookup n t of
  Just v   -> Ans_S v t
  Nothing -> Que_S n t (\v t' -> Ans_S v ((n, v) : t'))
```

The function `stree2fun` recovers the functional from its strategy tree representation.

```
stree2fun :: (s -> Tree_S s i o a) -> Func_S s i o a
stree2fun t f = get >>= go o t where
  go (Ans_S a s) = put s >> return a
  go (Que_S i s k) = put s >> f i >>= (\o -> get >>= go o k o)
stree2open :: StateM s m => (i -> s -> Tree_S s i o o) -> Open (i -> m o)
stree2open f super i = stree2fun (f i) super
```

For instance, we have the following relation between `memo` and `memot`:

$$\text{memo} \equiv \text{stree2open memot}$$

Unfortunately, this section's approach is far too ad-hoc. It is unclear how the stateful variants of `Tree` and `tree2fun` are actually derived, let alone how to do so for the `stree2fun`'s inverse or to adapt the definitions to other kinds of effects. Clearly, we need a more systematic and structured way to derive stateful variants of Bauer et al.'s pure definitions.

5 Strategy Trees as Coroutines

The first step towards a more structured approach is the observation that strategy trees are effectively *coroutines*. In functional programming, they are better known as the coroutine *monad*, aka the resumption monad.

```
instance Monad (Tree i o) where
  return      = Ans
  Ans x >>= f = f x
  Que i k >>= f = Que i (\o -> k o >>= f)
```

In the co-routine interpretation, the `Ans` constructor denotes a completed computation, while the `Que` constructor denotes a suspended computation that can be resumed. A useful primitive operation is suspension:

```
suspend :: i -> Tree i a a
suspend i = Que i Ans
```

which enables a very concise conversion from a pure functional to its strategy tree:

```
fun2tree' :: Func i o a → Tree i o a
fun2tree' f = f suspend
```

In words, the functional's monad type parameter `m` is instantiated to the coroutine monad and the function parameter is instantiated to the `suspend` primitive.

For example, the strategy tree for `fib`, derived as `fun2tree' fib`, can be written in monadic style:

```
fibt' :: Int → Tree Int Int Int
fibt' n | n < 2 = return n
fibt' n        = do f1 ← suspend (n - 1)
                  f2 ← suspend (n - 2)
                  return (f1 + f2)
```

6 Coroutines with Effects

The next step in our structured approach is to reconcile the coroutine monad with other side effects, like state for `memo`. The solution to combine two effects, co-routines and another one, is entirely standard: monad transformers.

Thus, the coroutine monad transformer replaces the coroutine monad. It is defined as follows:

```
newtype CorT i o m a = CorT {runCorT :: m (C i o m a)}
data C i o m a
    = Done a
    | Suspend i (o → CorT i o m a)
instance MonadT (CorT i o) where
    lift = CorT ∘ liftM Done
    tbind m f =
        CorT (do x ← runCorT m
               case x of
                   Done y      → runCorT $ f y
                   Suspend i k → return (Suspend i (λo → k o 'tbind' f)))
    susp :: Monad m ⇒ i → CorT i o m o
    susp i = CorT $ return (Suspend i return)
```

From Polymorphic to Monomorphic Form We proceed in a two steps:

1. Use the coroutine transformer to transform a higher-order monadic function into a first-order one. This happens by partially instantiating the monad type parameter `m` to `CorT n`, where `n` takes care of other side effects.¹

¹ Every `Monad` is also a `Functor`, though this is not explicit at the type class level. The `Functor` constraint is needed shortly.

```

step1 :: (∀m.(Functor m, Monad m) ⇒ (i → m o) → m a)
        → (∀n.Monad n ⇒ CorT i o n a)
step1 f = f susp

```

2. Eliminate the polymorphic type variable `n` by instantiating it to a concrete implementation. In the case the constraint on `n` is just `Monad n`, like for `fib`, the canonical instantiation is the identity monad – there are no side effects.

```

step2 :: (∀n.Monad n ⇒ CorT i o n a) → CorT i o Id a
step2 m = m

```

In summary, we obtain a monomorphic characterization as follows:

```

poly2mono :: (∀m.(Functor m, Monad m) ⇒ (i → m o) → m a)
            → CorT i o Id a
poly2mono m = step2 (step1 m)

```

For instance, here is the monomorphic characterization of the Fibonacci function:

```

fibt'' :: Int → CorT Int Int Id Int
fibt'' n = poly2mono (flip fib n)

```

From Monomorphic to Polymorphic Form The opposite transformation is similar two step process.

1. Replace the concrete monad implementation by a type parameter, by means of a monad morphism. In the example, the monad morphism `id2any` does this:

```

type m ~ n = ∀a.m a → n a
id2any :: ∀n.Monad n ⇒ Id ~ n
id2any = return ∘ runId

```

The morphism is applied through the `CorT` transformer by means of `Monatron`'s library function:

```

tmap :: (FMonadT t, Functor m, Functor n) ⇒ (m ~ n) → t m a → t n a

```

In summary, this step consists of:

```

step3 :: CorT i o Id a → (∀n.(Functor n, Monad n) ⇒ CorT i o n a)
step3 m = tmap id2any m

```

2. Eliminate the suspension transformer again, by means of:

```

step4 :: (∀m.(Functor m, Monad m) ⇒ CorT i o m a)
        → (∀m.(Functor m, Monad m) ⇒ (i → m o) → m a)

```

```

step4 m f = go m where
  go m = do x ← runCorT m
        case x of
          Done y      → return y
          Suspend i k → f i >>= go ∘ k

```

Hence, the reverse process is:

```

mono2poly :: CorT i o Id a → (∀m. (Functor m, Monad m) ⇒ (i → m o) → m a)
mono2poly m = step4 (step3 m)

```

Bringing everything together, we claim that `poly2mono` and `mono2poly` are each other's inverses:

```

mono2poly ∘ poly2mono ≡ id
poly2mono ∘ mono2poly ≡ id

```

which means that the monomorphic representation is isomorphic to the polymorphic one, and thus a useful characterization.

7 Coroutines with Algebraic Effects

This section adapts the two-step approach to two different side effects, state and non-determinism, that expose only *algebraic* operations. Jaskelioff [6] calls a monadic operation *op algebraic* if its type has the form $\forall a. F\ a \rightarrow M\ a$, where F is some functor. Algebraic operations are easily lifted with `lift ∘ op :: ∀a. F a → T M a`.

7.1 Stateful Coroutines

The only changes in `step1` and `step2` are the replacement of `Monad` by `StateM s`, and of `Id` by `State s`:

```

step1,S      :: (∀m. (Functor m, StateM s m) ⇒ (i → m o) → m a)
               → (∀n. StateM s n ⇒ CorT i o n a)
step1,S f = f susp
step2,S :: (∀n. StateM s n ⇒ CorT i o n a) → CorT i o (State s) a
step2,S m = m
poly2monoS :: (∀m. (Functor m, StateM s m) ⇒ (i → m o) → m a)
             → CorT i o (State s) a
poly2monoS m = step2,S (step1,S m)

```

Note that `step1,S` works because Monatron's infrastructure lifts the `get` and `put` operations from `n` to `CorT i o n`.

Now `poly2monoS` yields a monomorphically typed variant of `memo`:

```

memot'' :: Eq k => k -> CorT k v (State [(k,v)]) v
memot'' k = poly2monoS (flip memo k)

```

Note that if we specialize the type `CorT i o (State s) a`, we indeed obtain `s -> TreesS i o a`, i.e., the ad-hoc stateful strategy tree we postulated earlier.

In the other direction, `memo` is recovered from `memot''` by means of `step3,S` and `step4,S`, of which the former uses the `state2any` morphism and the latter differs from `step4` merely in its signature.

```

state2any :: StateM s m => State s ~> m
state2any m = do s0 <- get
              let (x, s1) = runState s0 m
              put s1
              return x

step3,S :: CorT i o (State s) a -> (forall n. (Functor n, StateM s n) => CorT i o n a)
step3,S m = tmap state2any m

step4,S :: (forall m. (Functor m, StateM s m) => CorT i o m a)
         -> (forall m. (Functor m, StateM s m) => (i -> m o) -> m a)
step4,S m f = go m where
  go m = do x <- runCorT m
          case x of
            Done y      -> return y
            Suspend i k -> f i >>= go o k

mono2polyS :: CorT i o (State s) a
            -> (forall m. (Functor m, StateM s m) => (i -> m o) -> m a)
mono2polyS m = step4,S (step3,S m)

```

We claim again that `poly2monoS` and `mono2polyS` are each other's inverses:

$$\begin{aligned} \text{mono2poly}_S \circ \text{poly2mono}_S &\equiv \text{id} \\ \text{poly2mono}_S \circ \text{mono2poly}_S &\equiv \text{id} \end{aligned}$$

The `WriterM` and `ReaderM` effects with their respective algebraic operations `tell` and `ask` are treated in a similar fashion.

7.2 Non-Deterministic Coroutines

Another interesting effect with algebraic operations is non-determinism, a control-flow effect. Monatron's type class for non-deterministic effects is `ListM`, which supplies operations `mZero :: ListM m => m a` and `mPlus :: ListM m => m a -> m a -> m a` for respectively no solutions and merging of solutions. The key ingredients we need for the approach are a canonical implementation of `ListM`, the list monad `[]`, and a morphism:

```

list2any :: ListM m => [] ~> m
list2any []      = mZero
list2any (x : xs) = return x 'mPlus' list2any xs

```

Following the two-step approach we obtain again two transformations poly2mono_L and mono2poly_L , which we claim are each other's inverses.

8 Non-Algebraic Operations

Unfortunately, our two-step approach based on CorT breaks down when handling non-algebraic operations. As an example we study exceptions, with type class $\text{ExcM } x$ and operations $\text{throw} :: \text{ExcM } m \Rightarrow x \rightarrow m \text{ a}$ and $\text{handle} :: \text{ExcM } x \Rightarrow m \text{ a} \rightarrow (x \rightarrow m \text{ a}) \rightarrow m \text{ a}$ for throwing and handling exceptions. While the throw operation is algebraic, the handle operation is not: it has a type of the form $\forall a. F (M \text{ a}) \rightarrow M \text{ a}$ rather than $\forall a. F \text{ a} \rightarrow M \text{ a}$.

8.1 The Problem of Exceptions and Coroutines

Following the approach, $\text{Exception } x^2$ serves as the canonical implementation of $\text{ExcM } x$ with the following morphism:

```
exc2any :: ∀x m. ExcM x m ⇒ Exception x ⇔ m
exc2any m = either2any (runException m)
either2any :: ∀x m. ExcM x m ⇒ Either x ⇔ m
either2any m = case m of
  Left x  → throw x
  Right a → return a
```

This results in the two operations:

```
poly2monoX :: (∀m. ExcM x m ⇒ (i → m o) → m a)
              → CorT i o (Exception x) a
mono2polyX :: CorT i o (Exception x) a
              → (∀m. ExcM x m ⇒ (i → m o) → m a)
```

which turn out not to be inverses because they do not preserve the semantics of handle . The following minimal example illustrates the issue:

```
f :: ∀m. ExcM () m ⇒ (() → m ()) → m ()
f g = g () 'handle' return
test1 = runException $ f (\_ → throw ())
test2 = runException $ mono2polyX (poly2monoX f) (\_ → throw ())

> test1
Right ()
> test2
Left ()
```

The reason is twofold:

² isomorphic to $\text{Either } x$

1. `CorT` externalizes the call to the function parameter `g`, which now happens in `mono2polyX`. When an exception is raised during a call, `mono2polyX` aborts and does not communicate it to the functional.
2. The definition of `handle`, uniformly lifted by `Monatron` from `Exception x` to `CorT i o (Exception x)`, exhibits the following property:

$$\text{handle } (m \gg (\lambda x \rightarrow \text{susp } i \gg f \ x)) \ h \equiv (\text{handle } m \ h) \gg (\lambda x \rightarrow \text{susp } i \gg f \ x)$$

In words, `susp` extracts itself and any subsequent computations from under `handle`. Hence, any exceptions that arise in `susp` or later, cannot be caught.

Of the two problems, the latter might be amended by customizing the lifting of `handle` for `CorT`. However, the former problem is more fundamental: `CorT` leaves no room for communicating errors back, which makes it altogether useless for our purpose.

8.2 Ad-hoc Solution

Fortunately, we can resort to a more ad-hoc solution again, and directly formulate a suitable monad that externalizes function calls, but preserves the desired semantics of `handle`.

```
data TreeX x i o a
  = ReturnX a
  | SuspendX i (Either x o → TreeX x i o a)
  | Raise x
```

This datatype has the two conventional constructors for: `ReturnX` for returning an answer immediatly and `SuspendX` for suspending the computation. Note that `SuspendX`'s continuation has a parameter of type `Either x o` rather than `o`. This way the external call can communicate its possible failure to the continuation. The `reifyX` function enables any external monad to reify its exception as an `Either x o` value

```
reifyX :: ExcM x m ⇒ m a → m (Either x a)
reifyX m = handle (m >> return ∘ Right) (return ∘ Left)
```

and `suspX` translates this communicated exception to the appropriate internal form:

```
suspX :: i → TreeX x i o o
suspX i = SuspendX i either2any
```

`Raise x` denotes `throw x`, a computation that results in exception `x`.

```
instance Monad (TreeX x i o) where
  return = ReturnX
  ReturnX a >> f = f a
```

$$\begin{aligned} \text{Suspend}_X i k \gg f &= \text{Suspend}_X i (\lambda o \rightarrow k o \gg f) \\ \text{Raise } x \gg f &= \text{Raise } x \end{aligned}$$

We spare the reader the particular infrastructure-related details of `ExcM` in `Monatron` and summarize `TreeX × i o`'s implementations of `throw` and `handle`.

As indicated above, `throw` is represented by `Raise`:

$$\text{throw } x = \text{Raise } x$$

The following two laws cover the `ReturnX` and `Raise` cases of `handle`:

$$\begin{aligned} \text{handle } (\text{return } a) h &\equiv \text{return } a \\ \text{handle } (\text{throw } x) h &\equiv h x \end{aligned}$$

while `SuspendX` defers `handle` to its continuation. Combined, we obtain the following definition of `handle`:

$$\begin{aligned} \text{handle } (\text{Return}_X a) h &= \text{Return}_X a \\ \text{handle } (\text{Suspend}_X i k) h &= \text{Suspend}_X i (\lambda o \rightarrow \text{handle } (k o) h) \\ \text{handle } (\text{Raise } x) h &= h x \end{aligned}$$

Now, the conversion from a functional to `TreeX` is essentially the same as before:

$$\begin{aligned} \text{fun2tree}_X &:: (\forall m. \text{ExcM } x m \Rightarrow (i \rightarrow m o) \rightarrow m a) \\ &\quad \rightarrow \text{Tree}_X x i o a \\ \text{fun2tree}_X f &= f \text{ susp}_X \end{aligned}$$

However, the inverse is more involved, interpreting the `TreeX` constructors in the target monad:

$$\begin{aligned} \text{tree2fun}_X &:: \forall m x i o a. \text{ExcM } x m \Rightarrow \text{Tree}_X x i o a \rightarrow ((i \rightarrow m o) \rightarrow m a) \\ \text{tree2fun}_X m f &= \text{go } m \text{ where} \\ \text{go} &:: \forall c. \text{Tree}_X x i o c \rightarrow m c \\ \text{go } (\text{Return}_X a) &= \text{return } a \\ \text{go } (\text{Suspend}_X i k) &= \text{reify}_X (f i) \gg \text{go } \circ k \\ \text{go } (\text{Raise } x) &= \text{throw } x \end{aligned}$$

This approach does preserve the semantics, as we can observe in the following example:

$$\text{test}_{2,X} = \text{runException } (\text{tree2fun}_X (\text{fun2tree}_X f) (_ \rightarrow \text{throw } ()))$$

and, in contrast to `test2`, we have:

$$\begin{aligned} &> \text{test}_{2,X} \\ &\text{Right } () \end{aligned}$$

Generally, we claim that `fun2treeX` and `fromXTree` are inverses.

9 Related Work

Monadic Reasoning Hutton [8] advises to remove the abstraction layer of operations like $\gg=$, `get` and `handle`, and to reason in terms of their concrete implementations. Unfortunately, when the monad is a polymorphic type variable, we cannot apply this approach directly. However, the aim in this paper is to obtain a monomorphic characterization of the monadic code first, in order to be able to reason about its concrete implementation.

Both of *parametricity* [9,10] and algebraic laws of monadic operations enable reasoning about polymorphic monads. Voigtländer [11] shows how to derive parametricity theorems for type constructor classes such as `Monad`. Many people [12,13] introduced and studied various algebraic laws to enable reasoning about monadic operations directly. Oliveira et al. [3] even combine parametricity and algebraic laws to reason about the non-interference of polymorphic monadic mixin components. However, their approach is not strong enough to establish the correctness of memoization, which relies on non-trivial invariants, in a modular fashion.

Unfortunately, the standard form of parametricity and algebraic laws are not powerful enough for our purpose due to the presence of the unknown function parameter. Hofmann et al. [4,5] perform custom logical relations reasoning to characterize pure higher-order functions in terms of *strategy trees*. This paper extends their work to impure higher-order functions in order to enable reasoning about polymorphic monadic mixin components.

The Coroutine Monad Different variants of the coroutine monad (transformer) [14] have been studied under different names: resumption monad [15], free monad [16] and step monad [17]. For our purposes the name *coroutine* is most suitable because it emphasizes that the computation is split into two parts, the internal part of the mixin component and the external part of the function parameter.

10 Conclusion

We have shown how to characterize polymorphic monadic mixin components as monomorphically typed first order definitions for a range of different side effects. For effects with only algebraic operations we provide a particularly systematic two-step approach based on the coroutine monad transformer. In future work, we aim to show how the monomorphic characterizations facilitate modular reasoning and reuse of proofs. Moreover, we will investigate how to extend the systematic approach to non-algebraic operation like exception handling.

References

1. Cook, W.R.: A Denotational Semantics of Inheritance. PhD thesis, Brown University. (1989)

2. Wadler, P.: Monads for functional programming. In: Program Design Calculi: Marktoberdorf International Summer School on Program Design Calculi. (1992)
3. Oliveira, B.C.d.S., Schrijvers, T., Cook, W.R.: EffectiveAdvice: disciplined advice with explicit effects. In: AOSD'10: 9th International Conference on Aspect-Oriented Software Development. pp. 109–120. (2010)
4. Hofmann, M., Karbyshev, A., Seidl, H.: What is a pure functional? In Abramsky, S., Gavioille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P., eds.: Automata, Languages and Programming. Volume 6199 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2010) pp. 199–210 10.1007/978-3-642-14162-1_17.
5. Bauer, A., Hofmann, M., Karbyshev, A.: On monadic parametricity of second-order functionals. (2012)
6. Jaskelioff, M.: Monatron: an extensible monad transformer library. In: Proceedings of the 20th international conference on Implementation and application of functional languages. IFL'08, Berlin, Heidelberg, pp. 233–248. Springer-Verlag (2011)
7. d. S. Oliveira, B.C., Schrijvers, T., Cook, W.R.: MRI: Modular Reasoning about Interference in Incremental Programming. <http://users.ugent.be/~tschrijv/Research/papers/mri.pdf> (2012)
8. Hutton, G., Fulger, D.: Reasoning About Effects: Seeing the Wood Through the Trees. In: Proceedings of the Symposium on Trends in Functional Programming, Nijmegen, The Netherlands. (2008)
9. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: IFIP Congress. pp. 513–523. (1983)
10. Wadler, P.: Theorems for free! In: FPLCA'89: 4th International Conference on Functional Programming and Computer Architecture. (1989) pp. 347–359
11. Voigtländer, J.: Free theorems involving type constructor classes. In: ICFP'09: 14th International Conference on Functional Programming. pp. 173–184. (2009)
12. Liang, S., Hudak, P.: Modular denotational semantics for compiler construction. In: ESOP'96: European Symposium on Programming. pp. 219–234. (1996)
13. Gibbons, J., Hinze, R.: Just do it: Simple monadic equational reasoning. In: ICFP'11: 16th International Conference on Functional Programming. pp. 2–14. (2011)
14. Blazevic, M.: monad-coroutine: Coroutine monad transformer for suspending and resuming monadic computations (2010) <http://hackage.haskell.org/package/monad-coroutine>.
15. Papaspyrou, N.S.: A Resumption Monad Transformer and its Applications in the Semantics of Concurrency. Technical Report CSD-SW-TR-2-01, National Technical University of Athens (2001)
16. Awodey, S.: Category Theory. Volume 49 of Oxford Logic Guides. Oxford University Press, Oxford (2006)
17. Jaskelioff, M., Moggi, E.: Monad transformers as monoid transformers. Theor. Comput. Sci. **411**(51-52), pp. 4441–4466 (December 2010)