Formal Verification of Monadic Computations

Steve Zdancewic

June 2024

1 Monad Laws

```
ret a >>= f === f a
m >>= ret === m
(m >>= f) >>= g === m >>= (\x -> g x >>= h)
```

These laws amount to saying that the Kliesli category has `ret` as a unit and has associative composition.

2 Properness

Bind must respect the monad equivalence

If $m_1 \approx m_2$, $k_1, k_2 : A \to MB$, and $\forall a_1, a_2 : A. a_1 = a_2 \Rightarrow k_1 a_1 \approx k_2 a_2$ then,

$$x m_1 k_1 x \approx x m_2 k_2 x$$

2.1 Properness for state

The notion of equivalence via $\approx$ for the state monad is function extensionality

$$s\ s \approx$$

$$s_1 s_2 k\ s_1\ s_2 \approx s_1 k\ s_1\ s_1$$

3 Extensible Semantics and the Free Monad

Toy language with just values and addition expressions. Have an interpreter

If we wanted to add a new expression, we could add a new inductive case for subtraction. Likewise, need to add an inductive case to the interpreter

This ad-hoc addition everywhere does not generalize well. Solution:

3.1 Datatypes a la carte

Expressions are now either values or *computations*. Index the computations by a type of operations, and give a function that turns a natural number into an operation expression

We then have an extensible method for adding things to our language

We can likewise have an extensible way to handle the interpreter by generally folding over an expression tree and providing case-wise methods for turning an operation into a number

Chain together sequential composition via continuations

After doing all the machinery to parametrize expressions, we can add in whatever subset of operations we wish. That is, given handlers (algebras) for Add and Minus, we can combine into a handler that supports both operations

4 Free Monads

Constructor for return

Cosntructor for `Do` that is again parametrized by a type of operations `Do {X} (op : E X) (k : X \to FFree E R)` Think of `E` as a type constructor. `E X` is a type of operations that gives back an `X`. i.e. index operation by return type

That is, the operations are dependently indexed by their return type and to be able to continue after returning and `X`, we need a continutation `k` that takes in an `X`

4.0.1 FFree Computations

Previously we defined a free monad `FFree` parameterized by a function `E : Type → Type` and a return type `R : Type`. It was noted that technically `FFree` is a freer monad rather than a free monad.

```
Inductive FFree (E : Type -> Type) (R : Type) : Type :=
| Ret (x : R)
| Do {X} (op : E X) (k : X -> FFree E R).
```

We now want to consider equivalence for `FFree E` computations. We do this through the equivalence relation `eq_FFree` which ensures that the continuations are extensionally equivalent.

```
Inductive eq_FFree {E X} : FFree E X -> FFree E X -> Prop :=
| eq_Ret : forall (x:X), eq_FFree (Ret x) (Ret x)
| eq_Do : forall {Y} (op : E Y) (k1 k2 : Y -> FFree E X)
  (Heq: forall (y1 y2:Y), y1 = y2 -> eq_FFree (k1 y1) (k2 y2)),
  eq_FFree (Do op k1) (Do op k2).
```

In addition to handling equivalence, we also want to be able to provide a notion of disjoint unions since our operation types are indexed by their return

types. We can do this by defining a sum type and convenient syntax for denoting that type.

```
Inductive sumi (E1 E2 : Type -> Type) (X : Type) : Type :=
| inli (_ : E1 X)
| inri (_ : E2 X).
Notation "Op1 +' Op2" := (sumi Op1 Op2) (at level 10).
```

We can also compose handlers using this sum type.

```
Definition hpure_sum {Op1 Op2} (h1 : forall X, Op1 X -> X) (h2 : forall X, Op2 X -> X)
: forall X, (Op1 +' Op2) X -> X :=
fun _ op => match op with
  | inli op => h1 _ op
  | inri op => h2 _ op
  end.
```

Finally, we can add in a generic trigger operation to inject an operation into our free monad. Trigger will be the building block for building up interesting nodes in our tree.

```
Definition trigger_ {E X} (e : E X) :=
  Do e (fun x => Ret x).
Notation trigger e := (trigger_ (inject e)).
```

With all this in place, we can use `fold_pure` as an interpreter to evaluate handlers. We can then write programs using free monads as shown below:

```
Example e1 : FFree (Plus +' BoolOp) nat :=
  b <- trigger (or true false);;
  if (b : bool)
  then trigger (add 10 10)
  else trigger (add 2 2).
Eval compute in fold_pure (hpure_sum hplus hbool) e1.
```