Monad translations compose

david@davidespinosa.net
October 2024

Monads and tree traversals

```
type 'a up      = 'a * int

type 'a down    = int -> 'a

type 'a through = int -> 'a * int
```

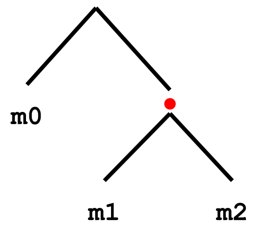
Prefix sum monad

```
module PrefixMonad (M : MONOID) =
  struct
    type 'a p = M.m -> 'a * M.m

    let returnP : 'a -> 'a p =
      fun a m0 -> (a, M.zero)

    let letP : 'a p -> ('a -> 'b p) -> 'b p =
      fun pa f m0 ->
      let (a, m1) = pa m0 in
      let (b, m2) = f a (M.add m0 m1) in
      (b, M.add m1 m2)
  end
```

```
(* Down : sum of values before
   Up   : sum of values below *)
```



Useful for hardware or neural nets ?

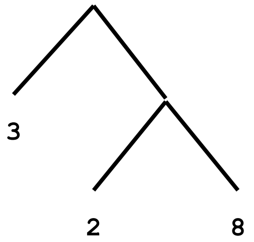
Tree sum

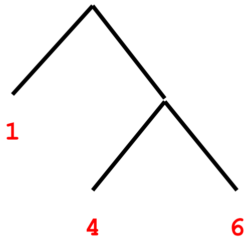
```
module PM = PrefixMonad(PlusMonoid)

type tree =
  | Leaf of int
  | Node of tree * tree

let rec tree_sum t =
  match t with
  | Leaf k ->
    fun s -> (Leaf s, k)
  | Node (l, r) ->
    PM.letP (tree_sum l) (fun l ->
      PM.letP (tree_sum r) (fun r ->
        PM.returnP (Node (l, r))))
```

```
tree_sum (Node (Leaf 2, Node (Leaf 5, Leaf 7))) 1
=> (Node (Leaf 1, Node (Leaf 4, Leaf 6)), 13)
```





Fast multiplication

```
let rec mul a n =
  if n = 0
  then PM.returnP ()
  else if n land 1 != 0
  then PM.letP (mul a (n - 1)) (fun () p -> ((), a))
  else PM.letP (mul a (n / 2)) (fun () p -> ((), p))
```

```
mul 2 15 0
=> ((), 30)
```

Exercise: Carry-lookahead adder

Monads don’t compose

joinST : STST → ST

returnS = insert S

returnT = insert T

joinS = combine SS → S

joinT = combine TT → T

An impossibility proof tells you what assumptions to violate ! 🙂

Monad translations

Start with an expression:

```
1 + 2
```

Monad translations

Monad translate it:

```
elet a = ereturn 1 in
  elet b = ereturn 2 in
    ereturn (a + b)
```

Monad translations

Inline the monad:

```
match (Some 1) with
  | None   -> None
  | Some a ->
    (match (Some 2) with
       | None   -> None
       | Some b -> Some (a + b))
```

Monad translations compose

Inline the monad:

```
match (Some 1) with
  | None    -> None
  | Some a ->
    (match (Some 2) with
      | None    -> None
      | Some b -> Some (a + b))
```

Repeat with another monad !

Language

Languages as theories / ADTs (Goguen et al)
Higher-order abstract syntax (Elliot, Pfenning)

```
module type LANG =
  sig
    type exp

    val unit    : exp
    val int     : int -> exp
    val string  : string -> exp

    val pair    : exp -> exp -> exp
    val left    : exp -> exp
    val right   : exp -> exp

    val inleft  : exp -> exp
    val inright : exp -> exp
    val case    : exp ->
                   (exp -> exp) ->
                   (exp -> exp) -> exp

    val elet : exp -> (exp -> exp) -> exp
    val lam  : (exp -> exp) -> exp
    val rlam : (exp -> exp -> exp) -> exp
    val app  : exp -> exp -> exp

    val nil  : exp
    val cons : exp -> exp -> exp
    val fold : exp -> exp -> exp -> exp

    val show : exp -> exp
    val ops  : (string * (bool * exp)) list
end
```

Monad

```
module type MONAD =
  sig
    type exp
    val ereturn : exp -> exp
    val elet    : exp -> (exp -> exp) -> exp
    val eshow   : exp -> exp
    val ops     : (string * (bool * exp)) list
  end
```

Monad

```
module type MONAD =
  sig
    type exp
    val ereturn : exp[a] -> exp[Ta]
    val elet    : exp[Ta] -> (exp[a] -> exp[Tb]) -> exp[Tb]
    val eshow   : exp -> exp
    val ops     : (string * (bool * exp)) list
  end
```

If we had dependent types, we could do a typed version.

Internal monad

```
module type IMONAD =
  functor (L : LANG) -> MONAD with type exp = L.exp
```

Option IMonad

```
module OptionIMonad(L : LANG) =
  struct
    type exp = L.exp

    (* a option = a + 1 *)

    let ereturn a =
      L.inleft a

    let elet ta f =
      L.case ta
        (fun a -> L.app (L.lam f) a)
        (fun x -> L.inright x)

    let eshow ta = ta
```

```
    let raise_op =
      L.lam (fun _ -> L.inright L.unit)

    let handle_op =
      L.lam (fun t1_t2 ->
          L.case (L.app (L.left t1_t2) L.unit)
            (fun a -> L.inleft a)
            (fun _ -> L.app (L.right t1_t2) L.unit))

    let ops = [
        ("raise", (false, ereturn raise_op));
        ("handle", (false, ereturn handle_op))
      ]
  end
```

Extend : IMONAD -> LANG -> LANG

Moggi, 1989

```
module Extend(IT : IMONAD)(L : LANG) =
  struct
    module T = IT(L)
    module X = Sugar(L)
    type exp(* a *) = L.exp(* Ta *)

    let lift1 f e =
      T.elet e (fun v -> T.ereturn (f v))

    let lift2 f e1 e2 =
      T.elet e1 (fun v1 ->
          T.elet e2 (fun v2 ->
              T.ereturn (f v1 v2)))

    let unit = T.ereturn L.unit
    let int k = T.ereturn (L.int k)
    let string s = T.ereturn (L.string s)

    let pair = lift2 L.pair
    let left = lift1 L.left
    let right = lift1 L.right
```

```
let inleft = lift1 L.inleft
let inright = lift1 L.inright

let case e f1 f2 =
  T.elet e (fun v ->
      L.case v
        (fun v1 -> f1 (T.ereturn v1))
        (fun v2 -> f2 (T.ereturn v2)))

let elet e f =
  T.elet e (fun v -> f (T.ereturn v))

let lam f =
  T.ereturn (L.lam (fun v -> f (T.ereturn v)))

let rlam f =
  T.ereturn (L.rlam (fun v1 v2 ->
      f (T.ereturn v1) (T.ereturn v2)))
```

Extend : IMONAD -> LANG -> LANG

Moggi, 1989

```
let app e1 e2 =
  T.elet e1 (fun v1 ->
      T.elet e2 (fun v2 ->
          L.app v1 v2))

let nil = T.ereturn L.nil
let cons = lift2 L.cons

let fold nil cons l =
  T.elet cons (fun cons ->
      T.elet l (fun l ->
          L.fold nil
            (L.lam (fun a ->
                 L.lam (fun sb ->
                     T.elet (L.app cons a)
                      (fun cons_a ->
                         T.elet sb (fun sb ->
                             L.app cons_a sb)))))
            l))
```

```
  let show e =
    L.show (T.eshow e)

  let lift_op (name, (ty, op)) =
    (name,
     (ty,
      T.ereturn
        (if ty
         then X.compose (L.lam T.ereturn) op
         else op)))

  let ops =
    List.append T.ops
      (List.map lift_op L.ops)
end
```

Monad translations compose

```
par
 (fun _ -> store 3)
 (fun _ -> store (1 + fetch ()))

[4; 1; 1; 3; 1; 1; 3; 1; 3; 3]
```

```
par
 (fun _ -> atomic (fun _ -> store 3))
 (fun _ -> atomic (fun _ -> store (1 + fetch ())))

[4; 3]
```

Example from:
Filinski, Representing Layered Monads

Monads: List, State, Resumptions

Monad translations compose

Espinosa: Monad translations compose !

Launchbury: Nice, but…

Monad translations compose

Espinosa: Monad translations compose !
Launchbury: Nice, but the result isn’t a monad translation.

```
Compose : IMONAD -> IMONAD -> IMONAD

Extend(T)(Extend(S)(L)) = Extend(Compose(T)(S))(L)
```

Compose : IMONAD -> IMONAD -> IMONAD

Possibly new !

```
module Compose(IT : IMONAD)(IS : IMONAD)(L : LANG) =
  struct
    module S   = IS(L)
    module LS  = Extend(IS)(L)
    module T   = IT(LS)
    module XLS = Sugar(LS)
    type exp = L.exp

    let ereturn a =
      T.ereturn (S.ereturn a)

    let elet sta f =
      T.elet sta (fun sa -> S.elet sa f)

    let eshow e =
      S.eshow (T.eshow e)
```

```
    let lift_op (name, (ty, op)) =
      (name,
       (ty,
        T.ereturn
          (if ty
           then XLS.compose (LS.lam T.ereturn) op
           else op)))

    let ops =
      List.append T.ops
        (List.map lift_op S.ops)
  end
```

ST.ereturn

```
ST.ereturn : a  -> STa

S.ereturn  : a  -> Sa
T.ereturn  : Sa -> STa   <= on Extend(S)(L)

let ST.ereturn a =
  T.ereturn (S.ereturn a)

Magic: Monad translate source code of T
```

ST.elet

```
ST.elet : STa -> (a  -> STb) -> STb

T.elet  : STa -> (Sa -> STb) -> STb   <= on Extend(S)(L)
S.elet  : Sa  -> (a  -> STb) -> STb

let ST.elet sta f =
  T.elet sta (fun sa -> S.elet sa f)

Magic: Monad translate source code of T
```

Compose(T) is a monad transformer

	`T`	`Compose(T)(S)`
`List`	`a list`	`S(a list)`
`Writer`	`a writer`	`S(a writer)`
`Option`	`a option`	`S(a option)`
`Reader`	`X -> a`	`S(X -> Sa)`
`State`	`X -> a * X`	`S(X -> S(a * X))`
`Continuation`	`(a -> R) -> R`	`S((a -> SR) -> SR)`
`Resumption`	`fix R. 1 -> a + R`	`S(fix R. 1 -> S(a + R))`

`Compose(T) : IMONAD -> IMONAD`

Proposition

Given:

- Monad S on ???
- Monad T on ???

There exists:

- Monad ST on ???

Proposition

Given:

Monad S on C

Monad T on Kleisli(S)

There exists:

Monad ST on C

Reference:

Barr and Wells,
Triples, Toposes, and Theories,
chapter on distributive laws.

Kleisli categories of monads on Kleisli categories

Asked 4 years, 11 months ago Modified 4 years, 11 months ago Viewed 213 times

5

In general, the composition of two monadic adjunctions is not necessarily itself monadic. Two known counterexamples include **TorsionFreeAb** → **Ab** → **Set** and **Cat** → **Quiver** → **Set** × **Set** (the latter is because **Cat**, the category of small categories, is not regular).

Now, let C be a category with a monad T and S be a monad on the Kleisli category C_T. Then, for any two objects X and Y in C, morphisms from X to Y in the Kleisli category $(C_T)_S$ correspond to morphisms from X to SY in C_T, which in turn correspond to morphisms from X to TSY in C. This suggests that $(C_T)_S$ is also the Kleisli category of a monad $\bar{S}$ on C for which $\bar{S}X = TSX$ on objects.

Question:

> Does there always actually exist such a monad $\bar{S}$? In other words, is the composition of two Kleisli adjunctions always itself a Kleisli adjunction?

category-theory

Share Cite Follow

asked Jul 12, 2019 at 17:18
Geoffrey Trang
10.1k ● 3 ■ 9 ▲ 34

Related (but doesn't answer) : you can find an answer, not for the Kleisli category but for the Eilenberg-Moore category, with the keywords distributive law - there is for instance an exercise sheet on Samuel Mimram's website about these : lix.polytechnique.fr/Labo/Samuel.Mimram/teaching/cat – Maxime Ramzi Jul 12, 2019 at 17:31

Add a comment

1 Answer

Sorted by: Highest score (default)

6

Surprisingly to me, yes-there's no need for any distributive laws or anything. If $U : D \leftrightarrows C : F$ is an adjunction, then by Peter Lumsdaine's answer here, our adjunction is equivalent to the Kleisli adjunction of the monad UF if and only if F is essentially surjective, and isomorphic to it if and only if F is bijective on objects. Both these properties are closed under composition of functors.

Share Cite Follow

answered Jul 12, 2019 at 18:42
Kevin Carlson
52.7k ● 4 ■ 60 ▲ 113

Characterization of Kleisli adjunctions

Asked 14 years ago Modified 13 years, 8 months ago Viewed 946 times

15

There's a well known theorem due to Beck that characterizes when an adjunction is monadic, that is, if F is left adjoint to G, $G : D \to C$, $GF := T$ is always a monad on C, and the adjunction is called monadic, essentially, when D is the Eilenberg–Moore category C^T of T-algebras and G is the forgetful functor. (For the precise definition see http://ncatlab.org/nlab/show/monadic+adjunction). I was wondering if there was a similar characterization to determine when D is the Kleisli category of FREE T-algebras?

ct.category-theory monads

Share Cite Improve this question Follow

edited Sep 29, 2010 at 16:35
Bjørn Kjos-Hanssen

asked May 26, 2010 at 22:03
David Carchedi

Add a comment

1 Answer

Sorted by: Highest score (default)

19

There is a unique functor $\mathbf{Kl}(GF) \to \mathbf{D}$ commuting with the adjunctions from $\mathbf{C}$, since the Kleisli category is initial among adjunctions inducing the given monad; and this functor is always full and faithful, since $\mathbf{Kl}(GF)(A, B) \cong \mathbf{C}(A, GFB) \cong \mathbf{D}(FA, FB)$.

So this functor will be an equivalence iff it is essentially surjective, and an isomorphism iff it is bijective on objects. But its object map is just the object map of F.

So $\mathbf{Kl}(FG)$ is equivalent to $\mathbf{D}$ compatibly with the adjunctions from $\mathbf{C}$ precisely when F is essentially surjective, and isomorphic just when F is bijective on objects.

Share Cite Improve this answer Follow

answered May 27, 2010 at 4:33
Peter LeFanu Lumsdaine

1 So, I guess this implies that if F is left adjoint to G and G does not reflect isos, then F cannot be essentially surjective? – David Carchedi May 27, 2010 at 6:28

Yep, I think so! More generally, G will always be full and faithful on the essential image of F, and hence reflect isomorphisms there. – Peter LeFanu Lumsdaine May 27, 2010 at 15:34

Compound Monads and the Kleisli Category

Jeremy E. Dawson *

Logic and Computation Program, NICTA ** and
Computer Sciences Laboratory,
Australian National University, Canberra, ACT 0200, Australia
`Jeremy.Dawson@nicta.com.au` `http://csl.anu.edu.au/~jeremy/`

2007

Abstract. We consider sets of monad rules derived by focussing on the Kleisli category of a monad, and from these we derive some constructions for compound monads. Under certain conditions these constructions correspond to a distributive law connecting the monads. We also show how these relate to some constructions for compound monads described previously.

Keywords: compound monad, Kleisli category

LetEff

```
module type LETEFF =
  sig
    include LANG

    val leteff :
      (exp[a]  -> exp[Ta]) ->
      (exp[Ta] -> (exp[a] -> exp[Tb]) -> exp[Tb]) ->
     exp[c] -> exp[c]
  end
```

```
Leteff appears in
Filinski, Monads in Action, 2010
with an operational semantics
```

LetEff

Possibly new !

```
module LetEff(L : LANG) =
  struct
    type exp = (L.exp -> L.exp) ->
     (L.exp -> (L.exp -> L.exp) -> L.exp) -> L.exp

    let int k sreturn slet =
      sreturn (L.int k)

    let leteff treturn tlet e sreturn slet =
      let streturn a =
        treturn (lift0 a) sreturn slet in
      let stlet sta f =
        slet sta (fun ta ->
            tlet (lift0 ta) (app (lift0 (L.lam f)))
              sreturn slet) in
      e streturn stlet

   ...
end
```

```
Note: We only use two layers:
  L0 = pure language, no effects
  L1 = LetEff(L0)
All other "layering" happens *inside* L1.
```

Async / await

Petricek and Syme,
F# Computation Expression Zoo, 2014

Async/await : one effect a time
LetEff : multiple effects at a time

```
leteff List   in
leteff Writer in
leteff Option in
...
```

Layered monads example

```
run
  (handle
     (elet (pick [1; 2; 3; 4; 5]) (fun a ->
          eseq (write ("a=" ^ (string_of_int a)))
            (eseq (if a * a = 9
                   then eseq (write "!") (eraise ())
                   else write "ok")
               (ereturn (10 * a)))))
     (eseq (write "H")
        (elet (pick [true; false]) (fun b ->
             if b
             then eseq (write "yes") (ereturn 42)
             else eraise ()))))
```

```
[(Some 10, "a=1ok");
 (Some 20, "a=2ok");
 (Some 42, "a=3!Hyes");
 (None,    "a=3!H");
 (Some 40, "a=4ok");
 (Some 50, "a=5ok")]
```

Example from:
Filinski, Monadic reflection in Haskell, 2006

Monads: List, Writer, Option

Implementation 0

```
type 'a exp = 'a option writer list
```

Implementation 1

```
type 'a exp0 = 'a
type 'a exp1 = 'a list   exp0
type 'a exp2 = 'a writer exp1
type 'a exp3 = 'a option exp2
```

```
(using monad transformers)
```

API

```
ereturn0 : 'a -> 'a exp0
ereturn1 : 'a -> 'a exp1
ereturn2 : 'a -> 'a exp2
ereturn3 : 'a -> 'a exp3

elet0    : 'a exp0 -> ('a -> 'b exp0) -> 'b exp0
elet1    : 'a exp1 -> ('a -> 'b exp1) -> 'b exp1
elet2    : 'a exp2 -> ('a -> 'b exp2) -> 'b exp2
elet3    : 'a exp3 -> ('a -> 'b exp3) -> 'b exp3
```

```
Filinski, Monadic reflection in Haskell, 2006
Filinski, Representing layered monads, POPL 1999
```

```
lift1     : 'a exp0 -> 'a exp1
lift2     : 'a exp1 -> 'a exp2
lift3     : 'a exp2 -> 'a exp3

reify1    : 'a exp1 -> 'a list   exp0
reify2    : 'a exp2 -> 'a writer exp1
reify3    : 'a exp3 -> 'a option exp2

reflect1 : 'a list   exp0 -> 'a exp1
reflect2 : 'a writer exp1 -> 'a exp2
reflect3 : 'a option exp2 -> 'a exp3

let run0 : 'a exp0 -> 'a

let run  : 'a exp3 -> 'a option writer list =
  fun e -> run0 (reify1 (reify2 (reify3 e)))
```

Implementation 2

```
type 'a exp0  = 'a
type 'a exp0k = { ka : 'r. ('a -> 'r exp0) -> 'r exp0 }

type 'a exp1  = 'a list   exp0k
type 'a exp1k = { ka : 'r. ('a -> 'r exp1) -> 'r exp1 }

type 'a exp2  = 'a writer exp1k
type 'a exp2k = { ka : 'r. ('a -> 'r exp2) -> 'r exp2 }

type 'a exp3  = 'a option exp2k
type 'a exp3k = { ka : 'r. ('a -> 'r exp3) -> 'r exp3 }
```

Implementation 3

```
type 'a exp0 = { ka : 'r. ('a -> 'r) -> 'r }
type 'a exp1 = { ka : 'r. ('a -> 'r list   exp0) -> 'r list   exp0 }
type 'a exp2 = { ka : 'r. ('a -> 'r writer exp1) -> 'r writer exp1 }
type 'a exp3 = { ka : 'r. ('a -> 'r option exp2) -> 'r option exp2 }
```

Implementation 4

```
type 'a exp0 = 'a k
type 'a exp1 = { ka : 'r. ('a -> 'r list   k) -> 'r list   k }
type 'a exp2 = { ka : 'r. ('a -> 'r writer k) -> 'r writer k }
type 'a exp3 = { ka : 'r. ('a -> 'r option k) -> 'r option k }
```

Continuation monads example

```
let returnK a k =
  k a

let letK ka f k =
  ka (fun a -> f a k)

let liftK ka =
  letK ka

let seqK e1 e2 =
  letK e1 (fun _ -> e2)

let appendK kla1 kla2 =
  letK kla1 (fun la1 ->
      letK kla2 (fun la2 ->
          returnK (la1 @ la2)))

let run0 ka =
  ka (fun a -> a)
```

```
let reify1 ka =
  ka (fun a -> returnK [a])

let reflect1 kla f =
  let rec loop la =
    match la with
    | [] -> returnK []
    | a :: la -> appendK (f a) (loop la) in
  letK kla loop

let reify2 ka =
  ka (fun a -> returnK (a, ""))

let reflect2 kwa f =
  letK kwa (fun (a, s1) ->
    letK (f a) (fun (b, s2) ->
      returnK (b, s1 ^ s2)))
```

Continuation monads example

```
let reify3 ka =
  ka (fun a -> returnK (Some a))

let reflect3 koa f =
  letK koa (fun oa ->
      match oa with
      | Some a -> f a
      | None   -> returnK None)

let pick l =
  liftK (liftK (reflect1 (returnK l)))

let write s =
  liftK (reflect2 (returnK ((), s)))
```

```
let eraise () =
  reflect3 (returnK None)

let handle e1 e2 =
  letK (liftK (reify3 e1)) (fun oa ->
      match oa with
      | Some a -> returnK a
      | None   -> e2)

let run e =
  run0 (reify1 (reify2 (reify3 e)))

let ereturn = returnK
let elet    = letK
let eseq    = seqK
```

Summary

Monads don’t compose, but…

Monad translations compose

Continuation monads compose

Also: Prefix sum monad 🙂

Turtles all the way

Brian Cantwell Smith, Reflection and semantics in a procedural language, 1982

Wand and Friedman, Mystery of the tower revealed, 1986

Why not an infinite tower ?

Challenge: Write a simulation that simulates itself
(literally – not a copy)

Consciousness

Challenge: Write the simplest conscious program

Morphisms of toposes

Logical relations / parametric polymorphism

Functions are related iff they take related arguments to related results.

Where do the relations come from ??

Possible answer: if we build the category of toposes as a subcategory of the category of allegories, it inherits morphisms from allegories.

Possible tutorials or textbooks

Automata theory via category theory

- Dexter Kozen, Automata and computability

Distributed systems

- Michel Raynal, 3 volumes (distributed, concurrent, fault tolerant)

Relaxed memory models (for concurrent shared memory)

- All papers inadequate

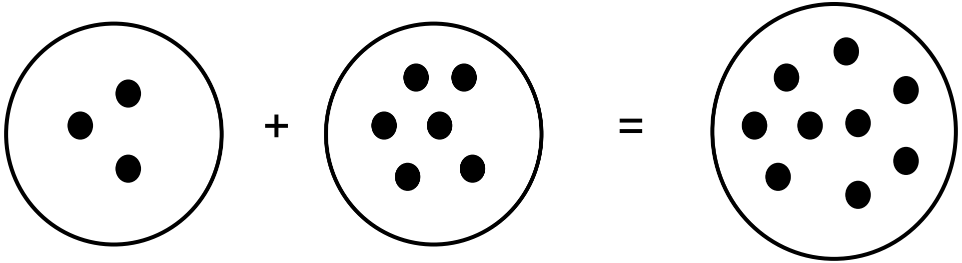
Relational algebra

- Bird and de Moor, Algebra of programming

Linear vs relational algebra

Linear algebra	Relational algebra
Apply linear transformation T(v)	??
??	Apply relation R(a, b)
Basis	??
Vector space	??
Well-known	Hardly known

Why is math “unreasonably effective” ?



Physics: Addition applies to marbles
Math: 3 + 5 = 8

Math is a collection of patterns.
Nature follows patterns…

Free will

Defn. A *controls* B = A correctly predicts actions of B

Defn. A *has free will* = A controls A

(standard confusion of causation vs correlation)

Something vs nothing

Why is there something rather than nothing ?

Occam’s razor fails !

Robert Kuhn, Closer to Truth