Monadic Reflection in Haskell

Andrzej Filinski
DIKU, University of Copenhagen, Denmark
andrzej@diku.dk

Mathematically Structured Functional Programming
Kuressaare, Estonia, July 2006
Motivation: two views of computational effects

Functional programmers approach effects in two qualitatively different ways:


  The light side: good and pure, but limiting.


  The dark side: powerful and fast, but dangerous.

Monadic reflection: a formal bridge between the two views.

Work so far: trying to get Scheme/ML programmers to see the light of monads.

Today: trying to lure Haskell programmers to the dark side (but just to pick up a few things...)
Plan

Start slow, hopefully end up somewhere non-trivial.

1. Motivating example: implementing output by state.

2. Implementing arbitrary monads by (composable) continuations, then by control-state behavior.

3. Implementing layered monad transformers; subeffecting.

Disclaimer 1: For presentation, focus on programming, not the underlying mathematical structures. Pretend that Haskell programs are their “obvious” denotations in domain theory, but details do need to be checked carefully, like for ML.

Disclaimer 2: Not very familiar with [contemporary] Haskell: may not use language features in optimal or most elegant way. Hope to advocate the main ideas, not their precise realization.
Reminder/notation: monads in Haskell

class Monad m where  -- m a = computations returning a-values
  return :: a -> m a
  >>= :: m a -> (a -> m b) -> m b  -- built-in strength
  -- ax1: return a >>= f = f a
  -- ax2: m >>= return = m
  -- ax3: (m >>= f) >>= g = m >>= \a -> f a >>= g

(What exactly does “=” mean in axioms? Roughly: denotational equivalence in PCF-like model, or observational equivalence w/o seq.)

Example: Error/exception monad:

data Maybe a = Just a | Nothing

instance Monad Maybe where
  return a = Just a
  m >>= f = case m of
    Just a -> f a
    Nothing -> Nothing
Part I: Implementing the output monad

Another example: (batch) output monad; no partial outputs observable.

```
newtype Output a = O { rO :: (a, String) } -- assumed atomic

instance Monad Output where
  return a = O (a,"")
  m >>= f = let (a, s) = rO m in
    let (b, s') = rO (f a) in
    O (b, s ++ s')

  -- monad laws follow from (String, ",", ++) being a monoid

Transparent definition: can freely define operators (both effect-introducing and -delimiting) wrt. representation, including:

out :: Char -> Output ()
out c = O ((), [c])

collect :: Output () -> String
collect m = snd (rO m)
```
Properties of Output-based characterization

Monad definition captures exactly possible behaviors of computation: returns a single result, and possibly-empty string output. Outputs from subcomputations concatenated in order.

Allows straightforward equational reasoning about output effects, e.g.:

```
length (collect (m1 >> m2))
= length (let (((),s1) = rO m1 in let (((),s2) = r0 m2 in s1++s2))
= let (((),s1) = r0 m1 in let (((),s2) = r0 m2 in length (s1++s2))
= let (((),s2) = r0 m2 in let (((),s1) = r0 m1 in length (s1++s2))
= let (((),s2) = r0 m2 in let (((),s1) = r0 m1 in length (s2++s1))
= length (collect (m2 >> m1))
```

But quite inefficient (worst-case quadratic). Easy fix: implement String monoid more efficiently (e.g., trees + flattening, or function-space monoid).

For illustration purposes, let’s do something more radical.
Another implementation of monadic output

Idea: maintain state of “output so far”, reversed for easy extension.
(In ML/Scheme: would probably keep as mutable cell, to avoid clutter.)

newtype Output' a = O' { rO' :: String -> (a, String) }

instance Monad Output' where
  return a = O' (\s -> (a, s))
  m >>= f = O' (\s -> let (a, s') = rO' m s in rO' (f a) s')
  -- this is just the String-state monad

out' :: Char -> Output' ()
out' c = O' (\s -> ((), c : s))

collect' :: Output' () -> String
collect' m = let ((), s) = rO' m [] in reverse s

Note that collect’ still returns a completely pure result.
Properties of Output’-based characterization

Pro: usually faster, especially if state monad implemented natively.

Con: collect’ has non-trivial cost: OK if used rarely.
Con: no guard against potentially undesirable behaviors:

\[
\text{flush} :: \text{Output'} () \quad \text{-- erase all output so far}
\]
\[
\text{flush} = 0' (\text{s} \rightarrow ((()), ""))
\]

\[
\text{peek} :: \text{Output'} \text{Char} \quad \text{-- return last char output}
\]
\[
\text{peek} = 0' (\text{s} \rightarrow (\text{head s}, \text{s}))
\]

Break simple abstraction of pure output-behavior. (If intentional, perhaps we really meant to implement a different abstraction.)

\[
\text{length} \ (\text{collect'} \ (\text{m1} >> \text{m2})) \neq \text{length} \ (\text{collect'} \ (\text{m2} >> \text{m1}))
\]
in general, though OK if m1 and m2 only use out’ and Output’-sequencing.

Can encapsulate \( (\text{Output'}, \text{return}, >>=, \text{out'}, \text{collect'}) \) as abstract type. But still non-trivial to formally show the equality above: it is only admissible, not derivable.
**Connecting Output and Output’**

Think of \( \text{Output} \) as values, \( \text{Output’} \) as behaviors. Want to relate them.

State-representation of a computation with output \( s \) consists of adding it (reversed) to accumulator:

\[
\text{reflect0} :: \text{Output} \ a \to \text{Output’} \ a \\
\text{reflect0} \ m = \text{let} \ (a, s) = \text{r0} \ m \ \text{in} \\
\quad (a, (\text{reverse} \ s \oplus s'))
\]

To determine computation output, run state-representation with empty accumulator and reverse:

\[
\text{reify0} :: \text{Output’} \ a \to \text{Output} \ a \\
\text{reify0} \ m' = \text{let} \ (a, s) = \text{r0'} \ m' \ \text{in} \ (a, \text{reverse} \ s)
\]

Principle of **monadic reflection**: encapsulate \( \text{Output’} \) as abstract type with \text{return}, \text{>>=}, \text{reflect0}, \text{reify0} as only operations.

Construct all other operators on \( \text{Output’} \) from \text{reflect0/reify0} and transparent definition of \( \text{Output} \).
Programming with reflect/reify

To get Output’-based version of operator, take Output-based definition, and replace uses of 0 with reflect0 . 0, and r0 with r0 . reify0:

```
out’ c = (reflect0 . 0) () ([c])
    = let (a, s) = r0 (0 () ([c])) in
        0’ (\s’ -> (a, (reverse s ++ s’)))
    = 0’ (\s’ -> (() , (reverse [c] ++ s’)))
    = 0’ (\s’ -> (() , c : s’)) -- just unfolding

collect’ m’ = snd ((r0 . reify0) m’)
    = snd (r0 (0 (let (a, s) = r0’ m’ []
                        in (a, reverse s))))
    = let (a, s) = r0’ m’ [] in reverse s
```

Easy to check that reify0 (reflect0 m) = m. But in general, still reflect0 (reify0 m’) ≠ m’, because might contain Output’-behaviors not expressible in Output. So have we achieved anything?

Yes, because will now show uniformly that reasoning about Output is sound for reasoning about Output’, assuming encapsulation.
Relating computations in Output and Output’

Relational approach. Unary: all typable Output’-terms are well-behaved.

Actually, goes through much smoother in binary formulation (Reynolds’74-style): all Output’-terms are related to their Output-counterparts.

Def. purification | · | on types and terms replaces all (Output’, return, >>=, reflect0, reify0) with (Output, return, >>=, id, id). Since Output’ was assumed abstract, purification preserves typability.

Want to show that complete program and its purification return identical results.

“In theory, there is no difference between theory and practice; in practice, there is.”

Proof sketch: define type-indexed relation between [denotations of] closed terms: for any type a, \((\sim_a) \subseteq \{t \mid \vdash t :: |a|\} \times \{t' \mid \vdash t' :: a\}\), where

\[
\begin{align*}
    s \sim_{\text{String}} s' & \iff s = s' \\
    p \sim_{(a,b)} p' & \iff \text{fst } p \sim_a \text{fst } p' \land \text{snd } p \sim_b \text{snd } p' \\
    f \sim_{a \rightarrow b} f' & \iff \forall a \sim_a a'. f a \sim_b f'a'
\end{align*}
\]
\[ m \sim_{\text{Output}} a m' \iff rO m \sim_{(a,\text{String})} rO m' \]

\[ m \sim_{\text{Output}'} a m' \iff rO' (\text{reflectO} m) \sim_{\text{String} \to (a,\text{String})} rO' m' \]

(With care, also extends to recursive types.)

**Lemma:** The operations of Output' are related to their purifications:

1. If \( a \sim a' \) then \texttt{return} \( a \sim_{\text{Output}'} a \texttt{return} a \).

2. If \( m \sim_{\text{Output}'} a m' \) and \( f \sim_{a \to \text{Output}'} f' \) then \( m >>= f \sim_{\text{Output}'} m' >>= f' \).

3. If \( m \sim_{\text{Output}} a m' \) then \texttt{id} \( m \sim_{\text{Output}'} \texttt{reflectO} m' \).

4. If \( m \sim_{\text{Output}'} a m' \) then \texttt{id} \( m \sim_{\text{Output}} \texttt{reifyO} m' \).

**Theorem:** If \( x_1 :: a_1, \ldots, x_n :: a_n \vdash t :: a \) and \( t_1 \sim_{a_1} t'_1, \ldots, t_n \sim_{a_n} t'_n \), then \( |t|[t_1/x_1, \ldots, t_n/x_n] \sim_a t[t'_1/x_1, \ldots, t'_n/x_n] \). Standard logical-relations proof, using lemma above.

**Corollary:** for closed \( \vdash p :: \text{String} \), \( |p| = p \).

**Corollary**\(^2\): if \(|t| = |t'|\) then \( t \equiv t' \) (obs.equiv.), because \(| \cdot | \) compositional.
Assessment

Monadic reflection provides a *lifeline* when venturing into behavioral effects: cannot express or observe anything not justifiable by the functional view.

Observational, but not denotational isomorphism. Actually, even a *monad isomorphism*: up to observation,

\[
\begin{align*}
\text{reflect}_0 (\text{return } a) &= \text{return } a \\
\text{reflect}_0 (m >>= f) &= \text{reflect}_0 m >>= (\text{reflect}_0 . f) \\
\text{reflect}_0 (\text{reify}_0 m) &= m \\
\text{reify}_0 (\text{return } a) &= \text{return } a \\
\text{reify}_0 (m >>= f) &= \text{reify}_0 m >>= (\text{reify}_0 . f) \\
\text{reify}_0 (\text{reflect}_0 m) &= m
\end{align*}
\]

But situation is *not* symmetric: also have all the usual constructors and reasoning principles for values of type `Output a`.

Note: could also have taken `Output' a` like `Output a`, but with more efficient implementation of `(String, ",", ++)`. Or make `Output'` continuation-based...
Part II: Implementing monads with continuations

Old observation, due to Wadler: continuations are as general as monads.

```
newtype Cont r a = C { rC :: ((a -> r) -> r) }

instance Monad (Cont r) where
    return a = C (\k -> k a)
    m >>= f = C (\k -> rC m (\a -> rC (f a) k))
```

With polymorphic continuations, very simple to implement any monad:

```
class Monad m => PCRmonad m where
    pcreflect :: m a -> Cont (m d) a
    pcreify :: (forall d. Cont (m d) a) -> m a

    pcreflect m = C (\k -> m >>= k)
    pcreify t = rC t return
```

Proof similar to before. For logical relation. \( \sim_{\text{Cont} \ d \ a} \) defined in terms of intersection of all possible relational interpretations of d.
Implementing output with continuations

How the continuation-based implementation works (omitting 0/r0):

Output’ a = \forall d. (a -> (d, String)) -> (d, String)

out’ c = pcreflect (((), [c])
        = C (\k -> (((), [c]) >>= k)
        = C (\k -> let (a, s) = (((), [c]) in
               let (b, s’) = k a in (b, s ++ s’))
        = C (\k -> let (b, s’) = k () in (b, [c] ++ s’))
        = C (\k -> let (b, s’) = k () in (b, c : s’))

Note: c is added in front of output. If k contains another out’-operation, it will come later in the list.

collect’ m = snd (pcreify m)
        = let (((), s) = rC m return in s
        = let (((), s) = rC m (\a -> (a, "")) in s

Initial continuation just returns results; sets up empty string for prepending. Functional data structure mimics reverse.
Making implementation a proper monad

Explicit polymorphism gets in the way. Need a typing dodge:

```haskell
import Data.Dynamic

fmDyn d = fromDyn d (error "Dynamic")

-- toDyn :: Typeable a => a -> Dynamic
-- fmDyn :: Typeable a => Dynamic -> a

Conceptually, all constructible types embeddable in universal type:

```haskell
data Dynamic = S String | F (Dynamic -> Dynamic) | ...
```

```haskell
class Monad m => CRmonad m where
    creflect :: m a -> Cont (m Dynamic) a
    creify :: Typeable a => Cont (m Dynamic) a -> m a
```

```haskell
creflect m = C (\k -> m >>= k) -- as before
creify t = rC t (\a -> return (toDyn a)) >>= (return . fmDyn)
    -- creify t = rC (unsafeCoerce t :: Cont (m a) a) return
```

In proof: all we need is that \( \text{fmDyn \cdot toDyn = id} \).
Implementing identity by continuations

Definitions of \( \text{cretefl} \)ect and \( \text{creify} \) used transparent definition of \( \text{Cont} \); actually models \textit{delimited continuations}: programmer-chosen result type.

Now, treat \( \text{Cont} \) as \textit{specification}, and implement it differently.

\textbf{Idea:} \( \text{Cont} \ r \ a = (a \to r) \to r = (a \to \text{Id} \ r) \to \text{Id} \ r \).

Implement \( \text{Id} \) as a monad of \textit{metacontinuations}, \( \text{Id}' \ r = (r \to d) \to d \):

\begin{verbatim}
  type Ans = Dynamic  -- not essential

  reflI :: a -> (a -> Ans) -> Ans
  reflI a = \k -> k a

  reifI :: Typeable a => ((a -> Ans) -> Ans) -> a
  reifI t = fmDyn (t (\a -> toDyn a))
  -- reifI t = unsafeCoerce t (\a -> a)

  So \( \text{Cont}' \ r \ a = (a \to (r \to d) \to d) \to (r \to d) \to d \).

  But this is actually nice to work with, if we take \( s = r \to d \).
\end{verbatim}
Embedding \textbf{Cont} in \textbf{ContState}

A monad of low-level behaviors: control and state, with \textit{fixed} type of answers:

\begin{verbatim}
newtype ContState s x = CS {rCS :: (x -> s -> Ans) -> s -> Ans}

instance Monad (ContState s) where  -- (looks just like Cont!)
  return a = CS (\k -> k a)
  m >>= f = CS (\k -> rCS m (\a -> rCS (f a) k))
\end{verbatim}

Can implement the monad \textbf{Cont} \( r \) as \textbf{Cont'} \( r \) = \textbf{ContState} \( \text{DCont} \ r \):

\begin{verbatim}
type DCont r = r -> Ans

reflK :: Typeable r => Cont r a -> ContState (DCont r) a
reflK m = CS (\k -> reflI (rC m (reifI . k)))

reifK :: Typeable r => ContState (DCont r) a -> Cont r a
reifK m = C (\k -> reifI (rCS m (reflI . k)))
\end{verbatim}

In Scheme or SML[/NJ], \textbf{ContState} is actually the language’s implicit effect monad: no programmer access to final answers, but can allocate cells in store.
Implementing monadic reflection

type Behavior s = ContState s
type MBeh m = DCont (m Dynamic)

class (Monad m, Typeable (m Dynamic)) => GRMonad m where
greflect :: m a -> Behavior (MBeh m) a
greify :: Typeable a -> Behavior (MBeh m) a -> m a

greflect m = reflK (C (\k -> m >>= k))
greify m = rC (reifK m) (\a -> return (toDyn a))
   >>= \d -> return (fmDyn d)

instance GRMonad []

test = greify (do a <- grefect [3::Int, 4] 
b <- grefect [5, 6] 
   return (a * b))
   -- test = [15,18,20,24]

Every *expressible* monad implementable with continuation-state behavior.
Part III: Implementing layered monads

So far: can implement any single monadic effect with Behavior.

What about combinations of effects? Cannot in general take any two existing monads and join them: not enough information.

Instead, monad transformers: parameterize monad definition by “base monad”, e.g., $T^\text{ex}_M A = M(A + 1)$, $T^r_M = R \to MA$. In general, monad of interest built up as chain of monad-transformer layers.

Want to define layered monadic reflection: one pair of operators per effect layer, not for entire monolithic monad.

Will now see true utility of ContState embedding: instead of using up entire state to implement a monad, use one cell per layer. Strategy:

1. Implement each monad layer in specification with a continuation layer.
2. Implement each continuation layer by one metacontinuation cell.

Will not go through the constructions in detail, but just show the code.
**Monad transformers in Haskell**

```haskell
class (Monad b, Monad (t b)) => MonadT t b where
  lift :: b a -> t b a -- monad morphism from b to t b
  -- ax1: lift (return a) = return a
  -- ax2: lift (m >>= f) = lift m >>= (lift . f)

Example: Maybe as a monad transformer:

```haskell
newtype MaybeT b a = MT { rMT :: b (Maybe a) }

```haskell
instance (Monad b) => Monad (MaybeT b) where
  return a = MT (return (Just a))
  b >>= f = MT (rMT b >>= \m -> case m of
                     Nothing -> return Nothing
                     Just a -> rMT (f a))

instance Monad b => MonadT MaybeT b where
  lift b = MT (b >>= \a -> return (Just a))
```
Monad layerings

Alternative presentation of relation between base and extended monad:

```haskell
class (Monad b, Monad (t b)) => MonadL t b where
    glue :: b (t b a) -> t b a -- struct. map of b-algebra t b a
    -- ax1: glue (return t) = t
    -- ax2: glue (b >>= f) = glue (b >>= (return . glue . f))
    -- ax3: glue b >>= f = glue (b >>= \t -> return (t >>= f))

instance Monad b => MonadL MaybeT b where
    glue b = MT (b >>= \t -> rMT t) -- ax1,2 automatically OK
```

Layering determine liftings (and vice versa):

```haskell
instance MonadL t b => MonadT t b where
    lift b = glue (b >>= \a -> return (return a))

instance MonadT t b => MonadL t b where
    glue m = lift m >>= \a -> a
```

Layerings technically more convenient; will recover lifting for behaviors.
Implementing layered continuations

-- type Void; empty :: Void -> a

escape :: ((a -> ContState s Void) -> ContState s Void) -> ContState s a
escape f = CS (∫k -> rCS (f (∫a -> CS (∫e -> k a))) empty)

incs :: ContState s x -> ContState (s, n) x
incs t = CS (∫k -> (∫(s, n) -> rCS t (∫a -> ∫s -> k a (s, n))) s)

type ExtDC s r = (s, s -> Ans) -- DCont r = ExtDC () r

tabort :: ContState s r -> ContState (ExtDC s r) Void
tabort t = CS (∫k -> (∫(s, mk) -> rCS t mk s)

vreset :: ContState (ExtDC s r) Void -> ContState s r
vreset t = CS (∫k -> (∫s -> rCS t empty (s, k))

lreflK :: Cont (ContState s r) a -> ContState (ExtDC s r) a
lreflK h = escape (∫k -> tabort (rC h (∫a -> (vreset (k a))))))

lreifK :: ContState (ExtDC s r) a -> Cont (ContState s r) a
lreifK t = C (∫k -> vreset (t >>= ∫a -> tabort (k a)))
Implementing layered monadic reflection

type Pure = ()
type ExtB s t = ExtDC s (t (Behavior s) Dynamic)

class MonadL t (Behavior s) => LRmonad s t where
  lreflect :: t (Behavior s) a -> Behavior (ExtB s t) a
  lreify :: Typeable a =>
    Behavior (ExtB s t) a -> t (Behavior s) a

  lreflect t = lreflK (C (\k -> return (t >>= (glue . k))))
  lreify t = glue (rC (lreifK t) (return . return . toDyn)
                  >>= \r -> return (r >>= (return . fm Dyn)))

inc :: Behavior s a -> Behavior (ExtB s t) a
inc = incs -- independent of t

run :: Typeable t => Behavior Pure t -> t
run t = fmDyn (rCS t (\a -> \() -> toDyn a) ())
Example: Output layer

newtype OutputT b a = OT { rOT :: b (a, String) }

instance Monad b => Monad (OutputT b) where
    return a = OT ((return (a, []))
    m >>= f = OT (rOT m >>= \(a, s) ->
                    rOT (f a) >>= \(b, s') -> return (b, s ++ s'))

instance Monad b => MonadL OutputT b where
    glue b = OT (b >>= \t -> rOT t)

instance LRmonad s OutputT

outs :: String -> Behavior (ExtB s OutputT) ()
outs s = lreflect (OT (((), s)))

runO :: Typeable a => Behavior (ExtB s OutputT) a
          -> Behavior s (a, String)
runO t = rOT (lreify t)
Example: Nondeterminism layer

newtype ListT b a = L { rL :: b [a] }

instance Monad b => Monad (ListT b) where
  return a = L (return [a])
  t >>= f = let mcf [] = return []
    mcf (h : t) = rL (f h) >>= \lh -> mcf t
    >>= \lt -> return (lh ++ lt)
  in L (rL t >>= \l -> mcf l)

instance Monad b => MonadL ListT where
  glue b = L (b >>= \t -> rL t)

instance LRmonad s ListT -- s must be commutative, e.g., Pure

pick :: [a] -> Behavior (ExtB s ListT) a
pick l = lreflect (L (return l))

runL :: Typeable a => Behavior (ExtB s ListT) a -> Behavior s [a]
runL t = rL (lreify t)
Example: Exception layer

```haskell
newtype MaybeT b a = MT { rMT :: b (Maybe a) }

instance Monad b => Monad (MaybeT b) -- as before
instance Monad b => MonadL MaybeT b -- as before

instance LRmonad s MaybeT

raise :: Behavior (ExtB s MaybeT) a
raise = lreflect (MT (return Nothing))

runM :: Typeable a => Behavior (ExtB s MaybeT) a
     -> Behavior s (Maybe a)
runM t = rMT (lreify t)

handle :: Typeable a => Behavior (ExtB s MaybeT) a
        -> Behavior (ExtB s MaybeT) a
        -> Behavior (ExtB s MaybeT) a
handle t1 t2 = inc (runM t1) >>= \m -> case m of
                      Just a -> return a
                      Nothing -> t2
```
Example: Environment (reader) layer

newtype ReaderT d b a = RT { rRT :: d -> b a }

instance Monad b => Monad (ReaderT d b) where
    return a = RT (\d -> return a)
    t >>= f = RT (\d -> rRT t d >>= \a -> rRT (f a) d)

instance Monad b => MonadL (ReaderT d) b where
    glue b = RT (\d -> b >>= \f -> rRT f d)

instance LRmonad s (EnvT d)

ask :: Behavior (ExtB s (ReaderT d)) d
ask = lreflect (RT (\d -> return d))

runR :: Typeable a => Behavior (ExtB s (ReaderT d)) a -> d -> Behavior s a
runR t = rRT (lreify t)

withd :: Typeable a => d -> Behavior (ExtB s (ReaderT d)) a -> Behavior (ExtB s (ReaderT d)) a
withd d t = inc (runR t d)
Example: programming with effect layers

silly = run (runL (run0 (runM
  (handle (do a <- inc (inc (pick [1..5]))
    inc (outs "a=" ++ show a))
    if a * a == 9 then do inc (outs "!"); raise
    else inc (outs "ok")
    return (10 * a :: Int))
  (do inc (outs "H")
    b <- inc (inc (pick [True,False]))
    if b then do inc (outs "yes"); return 42
    else raise))))

-- silly = [(Just 10,"a=1ok"),(Just 20,"a=2ok"),(Just 42,"a=3!Hyes"),
  (Nothing,"a=3!H"),(Just 40,"a=4ok"),(Just 50,"a=5ok")]

Note: complicated output type just for visualization purposes. All do-
computations actually live in Behavior monad

In practice would usually use effect-linking block to centralize level counting:

runAll = run . runL . run0 . runM
doPick = inc . inc . pick -- etc.
Data monads

Implementation paradigm: each layer of monadic effects represented by single metacontinuation cell in state.

Allows any monadic layer to be represented, including ones with control behaviors (ListT, MaybeT, ...)

But many monad layers involve no control behavior of their own: OutputT, ReaderT, StateT, ...; those can use their allotted state cells more directly.

General formulation: data monads generated by indexed monoids (like Output generated by monoid of strings). A whole other story, but fits nicely into general layering model:

- **Reasoning** bonus: all data monads commute with each other, so order of layering insignificant.

- **Efficiency** bonus: avoids one level of higher-order functions.
A final variation

Look at what parts of construction depend on details of \texttt{ContState}:

- State always has shape \((((), a1), \ldots, an)\).
- \texttt{tabort/vreset} access “top” cell of state, leave rest untouched.
- \texttt{incs} lifts computation to state with one additional cell
- \texttt{escape} is completely parametric: same for all instances of \texttt{ContState}.

Can play abstraction game once more: modify representation of state, change accessor operations.

- State is a dynamically extensible, flat \textit{store} with reference-indexed cells.
- \texttt{tabort/vreset} access their cell directly.
- \texttt{incs} is completely parametric in store shape: the identity function!
- \texttt{escape} must now save/restore parts of the store.

This is actually how the ML construction works: reflect/reify functions constructed wrt. fixed ordering of effects, cf. linking block.
Subtyping vs. subeffecting

Many languages support notion of subtyping: judgment $\tau \leq \tau'$; subsumption: if $t :: \tau$, then also $t :: \tau'$. Intuitively,

- $\tau'$ can add new elements to those already in $\tau$; variant subtyping: 
  $\text{(Red|Green)} \leq \text{(Red|Green|Blue)}$.
- $\tau'$ can identify previously distinct elements of $\tau$; record subtyping, 
  $\{x::\text{Real}, c::\text{Color}\} \leq \{x::\text{Real}\}$.

Modeled very intuitively by PERs: a type is a carrier set $+$ partial equivalence relation. Can assume that carrier set is fixed, e.g., natural numbers.

Similarly, can partially order available effects in language, $e \preceq e'$:

- Effect-embedding: supereffect adds new behaviors.
- Effect-projection: supereffect identifies behaviors (e.g. list vs. set).

Either super- or sub-effect can serve as implementation of specification. Must deal with unwanted elements / unwanted distinctions.
Inclusive vs. coercive subtyping/subeffecting

Subtyping can be semantically understood in two ways:

- coercive: subsumption involves a change of representation; \( \tau \leq \tau' \) determines a coercion function.
- inclusive: subsumption does not change representation; only interpretation of supertype is different.

Implementation of language with subtyping may use either or both.

Analogous situation for effects:

- Specification typically coercive (monad morphisms or layering): simplifies equational reasoning
- Implementation may be largely, or completely, inclusive; e.g., embedding everything in Behavior monad; type system ensures that meaningful.

More to multiple effects than just layered extensions/transformers.
Summary

Functionality from structure!

• **Motto**: effects are rare in functional programs; optimize for common case.

• A tiny bit of leeway (identity \(\rightarrow\) isomorphism) in writing monadic programs allows efficient implementation, without losing any reasoning precision.

• Any monad has efficient implementation in terms of \texttt{ContState}. So does any chain of monad transformers.

• Subeffecting may be coercive in specification, inclusive in implementation.

To be done…

• Work out convenient, idiomatic Haskell formulation of construction; ensure proper encapsulation, etc. Carry out larger-scale case study.

• Formalize & check all details (esp. strictness) in domain-theoretic setting \((M^3L)\); crucial, e.g., for working reliably with infinite streams.