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 "Haskell" view: Effects are *value* patterns: structuring tool for purely functional programs. Reasoning paradigm: *denotational*/*equational*. Typing: *Church*-style, effect-types came first.

  The light side: good and pure, but limiting.

- The "Scheme" view: Effects are *behavior* patterns: suitably constrained ways of using `set!` and `call/cc`. Reasoning paradigm: *operational*/*relational*. Typing (if any): *Curry*-style, effect-terms came first.

  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) = r0 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:

```
flush :: Output' ()   -- erase all output so far
flush = O' (\s -> ((), ""))

peek :: Output' Char  -- return last char output
peek = O' (\s -> (head s, s))
```

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

`length (collect' (m1 >> m2))` $\neq$ `length (collect' (m2 >> m1))` in general, though OK if `m1` and `m2` only use `out'` and `Output'`-sequencing.

Can encapsulate (`Output'`, `return`, `>>=`, `out'`, `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 `Output` as values, `Output'` as behaviors. Want to relate them.

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

```
reflectO :: Output a -> Output' a
reflectO m = let (a, s) = rO m in
               O' (\s' -> (a, (reverse s) ++ s'))
```

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

```
reifyO :: Output' a -> Output a
reifyO m' = O (let (a, s) = rO' m' [] in (a, reverse s))
```

Principle of **monadic reflection**: encapsulate `Output'` as abstract type with `return`, `>>=`, `reflectO`, `reifyO` as only operations.

Construct all other operators on `Output'` from `reflectO`/`reifyO` and transparent definition of `Output`.

Programming with reflect/reify

To get `Output'`-based version of operator, take `Output`-based definition, and replace uses of `O` with `reflectO . O`, and `rO` with `rO . reifyO`:

```
out' c = (reflectO . O) ((), [c])
       = let (a, s) = rO (O ((), [c])) in
           O' (\s' -> (a, (reverse s ++ s')))
       = O' (\s' -> ((), (reverse [c] ++ s')))
       = O' (\s' -> ((), c : s'))  -- just unfolding
collect' m' = snd ((rO . reifyO) m')
            = snd (rO (O (let (a, s) = rO' m' []
                          in (a, reverse s))))
            = let (a, s) = rO' m' [] in reverse s
```

Easy to check that `reifyO (reflectO m)` $=$ `m`. But in general, still `reflectO (reifyO m')` $\neq$ `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* $|\cdot|$ on types and terms replaces all (`Output'`, `return`, `>>=`, `reflectO`, `reifyO`) 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, $(\sim_{\mathtt{a}}) \subseteq \{t \mid\ \vdash t :: |\mathtt{a}|\} \times \{t' \mid\ \vdash t' :: \mathtt{a}\}$, where

$$
\begin{aligned}
s \sim_{\mathtt{String}} s' &\Leftrightarrow s = s' \\
p \sim_{\mathtt{(a,b)}} p' &\Leftrightarrow \mathtt{fst}\ p \sim_{\mathtt{a}} \mathtt{fst}\ p' \wedge \mathtt{snd}\ p \sim_{\mathtt{b}} \mathtt{snd}\ p' \\
f \sim_{\mathtt{a->b}} f' &\Leftrightarrow \forall a \sim_{\mathtt{a}} a'.\, f a \sim_{\mathtt{b}} f' a'
\end{aligned}
$$

$$m \sim_{\texttt{Output a}} m' \;\Leftrightarrow\; \texttt{rO}\ m \sim_{\texttt{(a,String)}} \texttt{rO}\ m'$$

$$m \sim_{\texttt{Output' a}} m' \;\Leftrightarrow\; \texttt{rO'}\ (\texttt{reflectO}\ m) \sim_{\texttt{String->(a,String)}} \texttt{rO'}\ m'$$

(With care, also extends to recursive types.)

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

1. If $a \sim_{\texttt{a}} a'$ then `return` $a \sim_{\texttt{Output' a}}$ `return` a.
2. If $m \sim_{\texttt{Output' a}} m'$ and $f \sim_{\texttt{a->Output' b}} f'$ then m `>>=` $f \sim_{\texttt{Output' b}} m'$ `>>=` f'.
3. If $m \sim_{\texttt{Output a}} m'$ then `id` $m \sim_{\texttt{Output' a}}$ `reflectO` m'.
4. If $m \sim_{\texttt{Output' a}} m'$ then `id` $m \sim_{\texttt{Output a}}$ `reifyO` m'.

Theorem: If $x_1 :: \mathsf{a}_1, \ldots, x_n :: \mathsf{a}_n \vdash t :: \mathsf{a}$ and $t_1 \sim_{\mathsf{a}_1} t'_i$, ..., $t_n \sim_{\mathsf{a}_n} t'_n$, then $|t|[t_1/x_1, \ldots, t_n/x_n] \sim_{\mathsf{a}} t[t'_1/x_1, \ldots, t'_n/x_n]$. Standard logical-relations proof, using lemma above.

Corollary: for closed $\vdash p :: \texttt{String}$, $|p| = p$.

Corollary^2: if $|t| = |t'|$ then $t \cong t'$ (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,

```
reflectO (return a)  =  return a
 reflectO (m >>= f)  =  reflectO m >>= (reflectO . f)
 reflectO (reifyO m)  =  m

   reifyO (return a)  =  return a
    reifyO (m >>= f)  =  reifyO m >>= (reifyO . f)
 reifyO (reflectO m)  =  m
```

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. $\sim_{\texttt{Cont d a}}$ defined in terms of *intersection* of all possible relational interpretations of d.

Implementing output with continuations

How the continuation-based implementation works (omitting `O`/`rO`):

```
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:

```
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:

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

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

  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 `fmDyn . toDyn` = `id`.

Implementing identity by continuations

Definitions of `creflect` and `creify` used transparent definition of `Cont`; actually models *delimited continuations*: programmer-chosen result type.

Now, treat `Cont` as *specification*, and implement it differently.

Idea: `Cont r a = (a -> r) -> r = (a -> Id r) -> Id r`.
Implement `Id` as a monad of *metacontinuations*, `Id' r` = `(r -> d) -> d`:

```
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 `Cont' r a = (a -> (r -> d) -> d) -> (r -> d) -> d`.

But this is actually nice to work with, if we take `s = r -> d`.

Embedding `Cont` in `ContState`

A monad of low-level behaviors: control and state, with *fixed* type of answers:

```
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))
```

Can implement the monad `Cont r` as `Cont' r = ContState (DCont r)`:

```
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)))
```

In Scheme or SML[/NJ], `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 <- greflect [3::Int, 4]
                  b <- greflect [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., $T^{\mathrm{ex}}_M A = M(A+1)$, $T^{\mathrm{r}}_M = R \to MA$. 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

```
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:

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

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:

```
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):

```
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, r -> 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 . fmDyn)))

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 (return ((), 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 b 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

```
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 (runO (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 . runO . 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 `ContState`:

- State always has shape `(((), a1), ..., an)`.
- `tabort`/`vreset` access "top" cell of state, leave rest untouched.
- `incs` lifts computation to state with one additional cell
- `escape` is completely parametric: same for all instances of `ContState` s

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

- State is a dynamically extensible, flat *store* with reference-indexed cells.
- `tabort`/`vreset` access their cell directly.
- `incs` is completely parametric in store shape: the identity function!
- `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,

- τ' can *add* new elements to those already in τ; *variant subtyping*: (Red|Green) $\leq$ (Red|Green|Blue).
- τ' can *identify* previously distinct elements of τ; *record subtyping*, {x::Real, c::Color} $\leq$ {x::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 `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^{3}L); crucial, e.g., for working reliably with infinite streams.