Dynamic instances

Albert ten Napel

```
-- Dynamic instances
-- First let's see why dynamic instances of effects are needed
-- Let's say we'd like to implement the following imperative algorithm
function f(b: boolean, input: int) {
  var x = input;
  if (b) {
    var y = x + 1;
    x = y * y;
  } else {
    x = x + 1;
  }
  return x;
}

-- Let's define the State effect
effect State t {
  get : () -> t
  set : t -> ()
}

-- with a standard state handler
-- with koka style parameterized handlers
runState : t -> r!<State t | e> -> r!e
runState = handle v {
  get () k -> k v v
  set v k -> k v ()
}

-- Now a possible implementation of f using the state effect
```

```
-- we need two state cells, so we use a pair
f : Bool -> Int -> Int
--                     (x    , y)
f b input = runState (input, 0) $ do
  (if b then do
    (x, y) <- get ();
    set (x, x + 1); -- var y = x + 1
    (_, y) <- get ();
    set (y * y, y) -- x = y * y
  else do
    (x, y) <- get ();
    set (x + 1, y)) -- x = x + 1
  (x, _) <- get ();
  return x

-- The state is encapsulated, but we have to give an initial value
-- for y even though we might not need y

-- so we want to actually use runState inside of the if
f' b input = runState input $ do
  (if b then do
    x <- get ();
    return $ runState (x + 1) $ do -- var y = x + 1
      y <- get ();
      set (y * y)) -- x = y * y
      -- Problem: we want to set x but the inner runState
      -- handles the set operation for y!

-- we actually want two state effects, so that
-- the operations don't interfere
effect State1 t {
  get1 : () -> t
  set1 : t -> ()
}
effect State2 t {
  get2 : () -> t
  set2 : t -> ()
}
```

```
-- with the obvious runStates
runState1 : t -> r!<State1 t | e> -> r!e
runState2 : t -> r!<State2 t | e> -> r!e

-- now we can implement the function properly
f'' b input = runState1 input $ do
  (if b then do
    x <- get1 ();
    return $ runState2 (x + 1) $ do -- var y = x + 1
      y <- get2 ();
      set1 (y * y) -- x = y * y
  else do
    x <- get1 ();
    set1 (x + 1)); -- x = x + 1
  x <- get1 ();
  return x

-- This works but requires us to know exactly how many
-- state values we want to manipulate

-- Now let's look at the following imperative function
-- that replaces each element in the list with the
-- greatest element that came before it (or itself)
function runningMax(list) {
  var max = 0;
  for(var i = 0; i < list.length; i++) {
    var cur = list[i];
    if(cur > max) {
      max = cur;
    } else {
      list[i] = max;
    }
  }
  return list;
}

-- Implementation in Haskell
-- (I could have used MArray, but I wanted to
```

```haskell
-- show the list of references being created)
runningMax :: [Int] -> ST s [Int]
runningMax l = do
  max <- newSTRef 0
  newl <- mapM newSTRef l
  runningMax' max newl 0
    where
      runningMax' max newl i = do
        if i >= (length newl) then
          mapM readSTRef newl
        else do
          cmax <- readSTRef max
          let cur = l !! i
          if cur > cmax then
            writeSTRef max cur
          else
            writeSTRef (newl !! i) cmax
          runningMax' max newl (i + 1)
-- Here we first create a list of new references
-- because we don't know the size of the list, we
-- also don't know how many references we are creating.
-- Furthermore we are looping through the list,
-- so we are also dynamically choosing which reference
-- to dereference.

-- similar to the state1 and state2 defined above,
-- we want state1 to staten, where n is the length of the list,
-- but since the length of the list is not statically known
-- we don't know how many state effects we need.
-- Also we are calling different references (operations) dynamically
-- So we don't know statically which state effect we are using.

-- This is where dynamic instances come in.
-- Similar to state1, state2 they create a new instance of the effect,
-- where the operations are called on the new instance, providing an
-- isolated version of the effect.
-- For the state effect an instance can be seen as a reference.
```

```
-- Now we can write the function f'' using instances
-- We have the following type constructors (in our hypothetical language)
State : Type -> Eff
Inst : Eff -> Type
Handler : Eff -> Type
-- And the following runState
-- (that takes an instance and a initial value as argument)
runState : Inst (State t) -> t -> t

fInst : Bool -> Int -> Int
fInst b input =
  let ref1 = new State in
  runState ref1 input $ do
  (if b then do
    x <- ref1#get ();
    let ref2 = new State;
    return $ runState ref2 (x + 1) $ do -- var y = x + 1
      y <- ref2#get ();
      ref1#set (y * y) -- x = y * y
  else do
    x <- ref1#get ();
    ref1#set (x + 1)); -- x = x + 1
  x <- ref1#get ();
  return x

-- References in this style are just new instances of the State effect
-- followed by a runState
-- Let's make a new effect that does this for us
New : Eff -> Eff
-- The New effect as one operation called new' which
-- takes an effect, a handler for that effect and returns
-- an instance of the effect
new' : (e:Eff) -> Handler e -> (Inst e)!<New e>
-- The default handler of new
handler handleNew {
  new' e h k ->
    let inst = new e in -- create a new instance of the give effect
    handle (k inst) with (h inst)
```

```
    -- run the continuation with the instance and handle with h
}

-- now we can define ref in terms of the New effect
-- ref gives the runState as the handler to wrap with to new'
ref : t -> Inst (State t)!<New (State t)>
ref v = new' (State t) (\inst -> runState inst v)

-- and we can define a function that uses refs
fRef : Int!<New (State Int)>
fRef = do
  r1 <- ref 1;
  r2 <- ref 2;
  x <- r1#get ();
  r2#set (x * 2);
  y <- r#set ();
  return (x + y)

-- we can run the state with handleNew
-- which will wrap every reference by a runState handler
result : Int
result = handleNew fRef
```