□ Type Renaming in Haskell

`newtype M = M Int`
creates a new type `M`, with an associated type constructor, which is isomorphic to `Int`. The underlying representation is equivalent to that for `Int`, the constructor `N` is just for the type checker.

`(\ (M x) -> 0) _|_` $\hookrightarrow$ `0`

`data N = N Int`
creates a new type `N`, with one constructor, which is really new. The underlying representation is different from that for `Int`.

`(\ (N x) -> 0) _|_` $\hookrightarrow$ `_|_`

□ Type Synonyms and Type Renaming

```
newtype M = M Int
```

```
type I = Int
```

Type synonyms (e.g. `I` above) are different from type renamings (e.g. `M` above).

- Type synonyms cannot be instances of a class, while type renamings can.
  `instance C I where ...` is not allowed.
  `instance C M where ...` is allowed.
- Type synonyms cannot be recursive while type renamings can.
  `type X = (Int, X)` is not allowed.
  `newtype X = X (Int, X)` is allowed.

□ Abstract Return Type for Interpreter

```
data Value = Wrong
           | Num Int
           | Fun (Value -> M Value)

unit :: Value -> M Value
bind :: M Value -> (Value -> M Value) -> M Value

getVar :: Env -> String -> M Value
add :: Value -> Value -> M Value
apply :: Value -> Value -> M Value

interp :: Term -> Env -> M Value
```

`M` is a type function we can define to get different interpreters.
Intuitively, `M Value` is a computation of a `Value`.

□ Interpreter Code

```
getVar :: Env -> String -> M Value
getVar ((x,v):xs) y = if x == y then unit v else getVar xs y
getVar [] y = bad $ "unbound variable "++y

add :: Value -> Value -> M Value
add (Num m) (Num n) = unit $ Num $ m + n
add m n = bad $ "expected numbers, found "++(show m)++" and "++(show n)

apply :: Value -> Value -> M Value
apply (Fun v1) v2 = v1 v2
apply f _ = bad $ "expected function, found "++(show f)

interp :: Term -> Env -> M Value
interp (Var x) env = getVar env x
interp (Con n) _ = unit $ Num n
interp (Add e1 e2) env =
    bind (interp e1 env) $ \v1 ->
    bind (interp e2 env) $ \v2 ->
    add v1 v2
interp (Lam x e) env = unit $ Fun $ \v -> interp e ((x,v):env)
interp (App e1 e2) env =
    bind (interp e1 env) $ \v1 ->
    bind (interp e2 env) $ \v2 ->
    apply v1 v2
```

□ Evaluation Strategy of Interpreter

- Should be call-by-value
  - `Fun :: Value -> M Value`
  - `Env = [(String, Value)]`
- Depending on `M`, might be call-by-value or call-by-need.
  - `type M a = a`
    Gives us call-by-need.
  - `data Id a = Id a`
    `type M a = Id a`
    Gives us call-by-value.
- We can switch to call-by-name.
  - `Fun :: M Value -> M Value`
  - `Env = [(String, M Value)]`

□ Interpreter Code (call-by-name)

```
getVar :: Env -> String -> M Value
getVar ((x,v):xs) y = if x == y then v else getVar xs y
getVar [] y = bad $ "unbound variable "++y

add :: Value -> Value -> M Value
add (Num m) (Num n) = unit $ Num $ m + n
add m n = bad $ "expected numbers, found "++(show m)++" and "++(show n)

apply :: Value -> M Value -> M Value
apply (Fun v1) v2 = v1 v2
apply f _ = bad $ "expected function, found "++(show f)

interp :: Term -> Env -> M Value
interp (Var x) env = getVar env x
interp (Con n) _ = unit $ Num n
interp (Add e1 e2) env =
    bind (interp e1 env) $ \v1 ->
    bind (interp e2 env) $ \v2 ->
    add v1 v2
interp (Lam x e) env = unit $ Fun $ \v -> interp e ((x,v):env)
interp (App e1 e2) env =
    bind (interp e1 env) $ \v1 ->
    apply v1 $ interp e2 env
```

□ Specific Versions of the Combinators

Original Interpreter

```
type M a = Id a
unit v = Id v
bad s = Id Wrong
bind (Id x) f = f x
```

Error Reporting Interpreter

```
type M a = Err a
unit v = Succ v
bad s = Err s
bind x f = case x of
  Succ v -> f v
  Err s -> bad s
```

□ Adding State

Add state (an `Int`) to keep track of computation steps.

```
data StateM a = StateM (Int -> (a, Int))
```

Use `data` to allow `instance` declarations later.

Combinators for interpreter with state:

```
unit v = StateM $ \s -> (v, s)

bad s = StateM $ \s -> (Wrong, s)

bind (StateM x) f = StateM $ \s0 ->
  let (v, s1) = x s0
      StateM next = f v
  in next s1
```

□ Some New Functionality

Add functions to manipulate state.

```
tick = StateM $ \s -> ((), s+1)
getSt = StateM $ \s -> (Num s, s)
```

Modify interpreter to record computation steps.

```
add (Num m) (Num n) = bind tick $ \_ -> unit $ Num $ m + n
apply (Fun v1) v2 = bind tick $ \_ -> v1 v2
```

Allow access to state in the language.

```
data Term = ... | Count
interp Count _ = getSt
```

□ Executing The Computation

`interp` now returns a `StateM Value` ≡ `StateM (Int -> (Value, Int))`

A `StateM` is a computation waiting for an initial state.

Since state is just for recording computation steps, the initial state is 0.

`show` function just provides the initial state.

```
instance Show a => Show (StateM a) where
    show (StateM m) =
        let (v, n) = m 0
        in "Value: "++(show v)++"\nSteps: "++(show n)++"\n"
```

□ Backwards State

Lazy evaluation allows us to do something strange.

```
bind (StateM x) f = StateM $ \s2 ->
    let (v1, s0) = x s1
        StateM next = f v1
        (v2, s1) = next s2
    in (v2, s0)
```

Effectively reverses flow of state through computation.

□ Adding Output

Add an output string to result value.

```
data Output a = Output (String, a)
```

Note difference from `StateM` in previous example.

```
unit v = Output ("", v)

bad s = Output (s, Wrong)

bind (Output (o1,v1)) f =
    let Output (o2, v2) = f v1
    in Output (o1++o2, v2)
```

Concrete output achieved if `bind` adds new output directly to screen.

□ Using Output

Allow values to be output.

```
data Term = ... | Out Term
```

```
out v = Output ((show v)++"; ", v)
```

```
interp (Out e) env = bind (interp e env) out
```

□ Non-Deterministic Programs

Allow multiple results.

```
unit v = [v]
bad s = [Wrong]
bind x f = concat $ map f x

both x y = x ++ y
```

Allow ambiguous terms.

```
Term = ... | Amb Term Term

interp (Amb e1 e2) env = both (interp e1 env) (interp e2 env)
```