monad-embed: a toy functional language

Tim Maxwell

March 16, 2011

```haskell
type Number = Double
data Term
    = Literal Number
    | Add Term Term
    | Compare Term Term
    | Cond Term Term Term

eval :: Term -> Number
eval = \ t -> case t of
    Literal d -> d
    Add a b -> eval a + eval b
    Compare a b ->
        if eval a == eval b then 1 else 0
    Cond a b c ->
        if eval a /= 0 then eval b else eval c
```

```
type Number = Double
data Term
    = Literal Number
    | Add Term Term
    | Compare Term Term
    | Cond Term Term Term
    | Alt Term Term
```

```
eval :: Term -> [Number]
eval = \ t -> case t of
    Literal d -> return d
    Add a b -> liftM2 (+) (eval a) (eval b)
    Compare a b -> do
        eq <- liftM2 (==) (eval a) (eval b)
        return (if eq then 1 else 0)
    Cond a b c -> do
        a' <- eval a
        if a' /= 0 then eval b else eval c
    Alt a b -> eval a ++ eval b
```

These two pieces of code are related.

```
if eval a == eval b then 1 else 0
```

```
do
    eq <- liftM2 (==) (eval a) (eval b)
    return (if eq then 1 else 0)
```

monad-embed looks like pure code.

```haskell
if eval a == eval b then 1 else 0
```

The monad-embed compiler transforms it into monadic code by inserting return and (>>=).

```haskell
do
    eq <- liftM2 (==) (eval a) (eval b)
    return (if eq then 1 else 0)
```

monad-embed looks like pure code.

```
case eval a `equal` eval b of {
     True -> 1; False -> 0;
     }
```

The monad-embed compiler transforms it into monadic code by inserting return and (>>=).

```
do
     eq <- liftM2 (==) (eval a) (eval b)
     return (if eq then 1 else 0)
```

```
eval :: (g :: * -> *) =>
    { ZList / g } (Term -> { ZList / g } Number);

eval = \ t -> case t of {
    Literal d -> d;
    Add a b -> eval a `plus` eval b;
    Compare a b ->
        case (eval a `equal` eval b) of
            { True -> 1; False -> 0; };
    Cond a b c ->
        case (eval a `equal` 0) of
            { False -> eval b; True -> eval c; };
    Alt a b -> either [eval a] [eval b];
    };
```

```
Int :: *
Maybe :: * -> *
Maybe Bool :: *
Either :: * -> (* -> *)
```

```
value :: Type
Type :: *
```

```
value :: {Flavor} Type;
Flavor :: * -> *    {- Flavor is a monad -}
Type :: *
```

```
value :: {Flavor} Type;
Flavor :: * -> *    {- Flavor is a monad -}
Type :: *
```

The flavor is the monad that provides the return and (>>=) operations that the compiler will insert.

monad-embed expressions have side effects; the flavor determines what the side effects can be.

monad-embed

Haskell

monad-embed

```
value :: {Flavor} Type;
```

Haskell

```
value :: Flavor Type
```

monad-embed	Haskell
`value :: {Flavor} Type;`	`value :: Flavor Type`
`A -> {F} R`	`A -> F R`

`f x`	**do**
	`f' <- f`
	`x' <- x`
	`f' x'`

`\ a -> a`	`return (\ a -> return a)`

Side effects are evaluated from left to right.

Function parameters are evaluated at the call site. Using the parameter within the function has no side effects. Defining a function has no side effects.

monad-embed	Haskell
`True`	`return True`
`Just`	`return $ \ a -> return (Just a))`
`case x of { True -> ...; }`	`do x' <- x; case x' of True -> ...`

```
True
```

```
return True
```

```
Just
```

```
return $ \ a ->
    return (Just a))
```

```
case x of {
    True -> ...;
    }
```

```
do
    x' <- x
    case x' of
        True -> ...
```

Constructors have no side effects.

case-expressions first perform the side effects of their parameter.

monad-embed

```
not :: (f :: * -> *) =>
    {f} (Bool ->
    {f} Bool);

not = \ x ->

    case x of {
        True ->
            False;
        False ->
            True;
        };
```

Haskell

```
not ::
    f (Bool ->
    f Bool)

not = return $ \ x -> do
    x' <- return x
    case x' of
        True ->
            return False
        False ->
            return True
```

```
eval :: (g :: * -> *) =>
    { ZList / g } (Term -> { ZList / g } Number);

eval = \ t -> case t of {
    Literal d -> d;
    Add a b -> eval a `plus` eval b;
    Compare a b ->
        case (eval a `equal` eval b) of
            { True -> 1; False -> 0; };
    Cond a b c ->
        case (eval a `equal` 0) of
            { False -> eval b; True -> eval c; };
    Alt a b -> either [eval a] [eval b];
    };
```

"ZList / g" means "The monad formed by composing the user-defined 'ZList' monad transformer with some monad, 'g'"

(monad-embed monad transformers aren't the same as Haskell monad transformer, but that's only important if you want to define your own.)

"ZList / g" means "The monad formed by composing the user-defined 'ZList' monad transformer with some monad, 'g'"

(monad-embed monad transformers aren't the same as Haskell monad transformer, but that's only important if you want to define your own.)

"eval :: (g :: * -> *) =>{ZList / g} ..." means that eval works with any stack of monad transformers, as long as the top one is ZList.

```
eval = \ t -> case t of {
    ...
    Add a b -> eval a `plus` eval b
    ...
    };

plus :: (f :: * -> *) =>
    {f} (Number ->
        {f} (Number ->
            {f} Number));
```

"plus" is parameterized on an arbitrary flavor. In the "eval" function, type inference determines that "f" is "ZList / g".

Almost all functions in monad-embed are parameterized on a monad.

Side-effect-free functions, like "plus", can have any flavor at all. This allows them to be used in any context.

```
eval :: (g :: * -> *) =>
    { ZList / g } (Term -> { ZList / g } Number);

eval = \ t -> case t of {
    Literal d -> d;
    Add a b -> eval a `plus` eval b;
    Compare a b ->
        case (eval a `equal` eval b) of
            { True -> 1; False -> 0; };
    Cond a b c ->
        case (eval a `equal` 0) of
            { False -> eval b; True -> eval c; };
    Alt a b -> either [eval a] [eval b];
    };
```

```
either :: (f :: * -> *, a :: *) =>
    {ZList / f} ([a] ->
        {ZList / f} [a] ->
            {ZList / f} a);

either = \ [x] [y] -> wrap [
    zcat
        (unwrap [x])
        [unwrap [y]]
    ];
```

```
unwrap :: (
        a :: *,
        f :: * -> *,
        t :: (* -> *) -> * -> *
    ) =>
    {f} [{t / f} a] -> {f} t f a;

x :: {ZList / f} Number;
unwrap [x] :: {f} ZList f Number;
```

"**unwrap**" lets us explicitly see side effects.

```
wrap :: (
        a :: *,
        f :: * -> *,
        t :: (* -> *) -> * -> *
    ) =>
    {t / f} [{f} t f a] -> {t / f} a;

zcat ... :: {f} ZList f Number;
wrap [zcat ...] :: {ZList / f} Number;
```

"**wrap**" lets us explicitly create side effects.

```
either :: (f :: * -> *, a :: *) =>
    {ZList / f} ([a] ->
        {ZList / f} [a] ->
            {ZList / f} a);

either = \ [x] [y] -> wrap [
    zcat
        (unwrap [x])
        [unwrap [y]]
    ];
```

"**wrap**" and "**unwrap**" are used to explicitly access the implicit monad.

```
eval :: (g :: * -> *) =>
    { ZList / g } (Term -> { ZList / g } Number);

eval = \ t -> case t of {
    Literal d -> d;
    Add a b -> eval a `plus` eval b;
    Compare a b ->
        case (eval a `equal` eval b) of
            { True -> 1; False -> 0; };
    Cond a b c ->
        case (eval a `equal` 0) of
            { False -> eval b; True -> eval c; };
    Alt a b -> either [eval a] [eval b];
    };
```

```
main :: {IO} Unit;
main = output "Hello,␣World!";
```

monad-embed lets us create impure "mini-dialects" of Haskell.

```
data ErrorT (m :: *) (f :: * -> *) (a :: *) = {
    Success a;
    Failure m;
    };

transformer ErrorT (m :: *) where {
    return = Success;
    bind = \ a f -> case a of {
        Success x -> f x;
        Failure m -> Failure m;
        };
    };

fail :: (m :: *, f :: * -> *, a :: *) =>
    {ErrorT m / f} m -> a;
fail = \ m -> wrap [Failure m];
```

```
divide' :: (g :: * -> *) => {ErrorT String / g}
    Number -> Number -> Number;

divide' = \ x y -> case y `equal` 0 of {
    True -> fail "Zero division.";
    False -> x `divide` y;
    };

main = do {
    x = strToNum input;
    y = strToNum input;
    res = unwrap [divide' x y];
    } then case res of {
        Success d -> output (numToStr d);
        Failure m -> output m;
        };
```

```
Input: 2
Input: 5
Output: 0.4

Input: 2
Input: 0
Output: Zero division.
```

monad-embed makes it easier to reuse higher-order functions.

```
main = do {
    numbers = 1 `Cons` 2 `Cons` 3 `Cons` Nil;
    numbers' = filter askUser numbers;
    } then outputNumbers numbers';

askUser :: () => {IO} (Number -> {IO} Bool);
askUser = \ question -> do {
    output ("Classify␣" `strCat`
        numToStr question `strCat`
        "␣(y/n)");
    answer = input;
    } then (answer `strEqual` "y");
```

```
filter = \ f l -> case l of {
    Nil -> Nil;
    Cons x xs -> case f x of {
        True -> x `Cons` filter f xs;
        False -> filter f xs;
        };
    };
```

```
Output: Classify 1.0 (y/n)
Input: y
Output: Classify 2.0 (y/n)
Input: n
Output: Classify 3.0 (y/n)
Input: y
Output: 1.0
Output: 3.0
```

```
main = do {
    numbers = 1 `Cons` 2 `Cons` 3 `Cons` Nil;
    numbers' = filter askUser numbers;
    } then outputNumbers numbers';

askUser :: () => {IO} (Number -> {IO} Bool);
askUser = \ question -> do {
    output ("Classify " `strCat`
        numToStr question `strCat`
        " (y/n)");
    answer = input;
    } then (answer `strEqual` "y");
```

filter 's author didn't have to know we would use filter like this.

This doesn't illustrate a point; I just think it's cool.

```
plusOrMinus = either [plus] [minus];

quadraticFormula = \ a b c ->
    (negate b `plusOrMinus` sqrt (
        (b `times` b) `minus`
        (4 `times` a `times` c)
    ))
    `divide` (2 `times` a);
```

The End.

http://timmaxwell.org/pages/monad-embed/