Little Self-Replicating Programs

Alex Gajewski (apg2162)

December 18, 2019

1 Value

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
{-# LANGUAGE MultiParamTypeClasses #-}

module Value (
    Value(..),
    EvalError(..),
    Thread(..),
    WorldState(..),
    throw,
    pause,
    runThread,
    liftRandom,
) where

import Control.Monad.Identity
import Control.Monad.Except
import Control.Monad.State
import Control.Monad.Coroutine

import Control.Monad.Random
import System.Random

import qualified Data.Map as Map

data Value = IntVal Int
           | PrimFunc String (Value -> Thread Value)
           | Lambda Int Value
           | Variable Int
           | FuncCall Value Value

instance Show Value where
    show (IntVal x) = show x
    show (PrimFunc name _) = name
    show (Lambda var val) = "(lambda var:" ++ show var ++ " " ++ show val ++ ")"
    show (Variable var) = "var:" ++ show var
    show (FuncCall f a) = "(" ++ show f ++ " " ++ show a ++ ")"

type ValueMap = Map.Map Int Value

data EvalError = EvalError

data WorldState = WorldState { univMap :: ValueMap,
                               univSize :: Int,
                               univEdits :: ValueMap,
                               envMap :: ValueMap,
                               randomGen :: StdGen,
                               cellPos :: Int }
                deriving (Show)
```

```
newtype Thread a =
    Thread (Coroutine Identity (ExceptT EvalError (StateT WorldState Identity)) a)
    deriving (Functor,
              Applicative,
              Monad)

instance MonadState WorldState Thread where
    get = Thread $ lift $ get
    put = Thread . lift . put

throw :: EvalError -> Thread a
throw = Thread . lift . throwError

pause :: Thread ()
pause = Thread $ suspend $ Identity $ return ()

type Unwrapped a = (Either EvalError (Either (Thread a) a), WorldState)

runThread :: WorldState -> Thread a -> Unwrapped a
runThread state (Thread t) =
    unwrapId . runIdentity . flip runStateT state . runExceptT . resume $ t
    where
        unwrapId (Right (Left (Identity t)), s) = (Right $ Left $ Thread t, s)
        unwrapId (Right (Right x), s) = (Right $ Right x, s)
        unwrapId (Left err, s) = (Left err, s)

liftRandom :: Rand StdGen a -> Thread a
liftRandom rand = do
    state <- get
    let (x, g) = runRand rand $ randomGen state
    put $ state { randomGen = g }
    return x
```