Little Self-Replicating Programs

Alex Gajewski (apg2162)

December 18, 2019

1 Value.lhs

```
{-# 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
```

2 State.lhs

```
module State (
    getVar,
    setVar,
    getCell,
    setCell,
    getCellPos,
    setCellPos,
    getSize,
) where

import Value

import Control.Monad.State

import System.Random

import qualified Data.Map as Map

getVar :: Int → Thread Value
getVar x = do
    state ← get
    case envMap state Map.!? x of
        Just y → return y
        Nothing → throw EvalError

setVar :: Int → Value → Thread ()
setVar x v = do
    state ← get
    put $ state { envMap = Map.insert x v $ envMap state }

getCell :: Int → Thread Value
getCell x = do
    state ← get
    return $ univMap state Map.! x

setCell :: Int → Value → Thread ()
setCell x v = do
    state ← get
    put $ state { univMap = Map.insert x v $ univMap state }

getCellPos :: Thread Int
getCellPos = do
    state ← get
    return $ cellPos state

setCellPos :: Int → Thread ()
setCellPos x = do
    state ← get
    put $ state { cellPos = x }

getSize :: Thread Int
getSize = do
    state ← get
    return $ univSize state
```

3 Eval.lhs

```
module Eval (
    eval,
) where

import Value
import State

eval :: Value → Thread Value
eval x@(IntVal _) = return x
eval x@(PrimFunc _ _) = return x
eval x@(Lambda _ _) = return x
eval (Variable x) = getVar x
eval (FuncCall f a) = do
    f' ← eval f
    case f' of
        PrimFunc _ g → do
            y ← g a
            pause
            return y
        Lambda x v → do
            setVar x a
            y ← eval v
            pause
            return y
        _ → throw EvalError
```

4 Builtins.lhs

```
module Builtins (
    primFuncs,
) where

import Value
import State
import Eval

primFuncs :: [Value]
primFuncs = [macro3 "if" ifFunc,
             macro2 "define" define,

             func1 "peek" peek,
             func2 "poke" poke,

             func2 "+" $ intOp (+),
             func2 "-" $ intOp (-),
             func2 "*" $ intOp (*),

             func2 ">" $ intBoolOp (>),
             func2 "<" $ intBoolOp (<),
             func2 "=" $ intBoolOp (==),

             func2 "&&" $ boolOp (&&),
             func2 "||" $ boolOp (||),

             func1 "eval" eval,

             func1 "lambda-get-var" lambdaGetVar,
             func1 "lambda-get-val" lambdaGetVal,
             func2 "lambda-set-var" lambdaSetVar,
             func2 "lambda-set-val" lambdaSetVal,

             func1 "funccall-get-func" funcCallGetFunc,
             func1 "funccall-get-arg" funcCallGetArg,
             func2 "funccall-set-func" funcCallSetFunc,
             func2 "funccall-set-arg" funcCallSetArg]

func1 :: String → (Value → Thread Value) → Value
func1 name f = PrimFunc name $ λx → do
               x' ← eval x
               f x'

func2 :: String → (Value → Value → Thread Value) → Value
func2 name f = PrimFunc name $ λx → return $
               PrimFunc (name ++ "1") $ λy → do
               x' ← eval x
               y' ← eval y
               f x' y'

macro2 :: String → (Value → Value → Thread Value) → Value
macro2 name f = PrimFunc name $ λx → return $
```

```
                PrimFunc (name ++ "1") $ λy →
                f x y

macro3 :: String → (Value → Value → Value → Thread Value) → Value
macro3 name f = PrimFunc name $ λx → return $
                PrimFunc (name ++ "1") $ λy → return $
                PrimFunc (name ++ "2") $ λz →
                f x y z

ifFunc :: Value → Value → Value → Thread Value
ifFunc b thenExpr elseExpr = do
    b' ← eval b
    case b' of
        IntVal x → if x > 0
            then eval thenExpr
            else eval elseExpr
        _ → throw EvalError

define :: Value → Value → Thread Value
define (Variable x) y = do
    y' ← eval y
    setVar x y'
    return y'
define _ _ = throw EvalError

peek :: Value → Thread Value
peek (IntVal x) = do
    n ← getSize
    y ← getCellPos
    getCell ((x + y) `mod` n)
peek _ = throw EvalError

poke :: Value → Value → Thread Value
poke (IntVal x) val = do
    y ← getCellPos
    n ← getSize
    setCell ((x + y) `mod` n) val
    return val
poke _ _ = throw EvalError

intOp :: (Int → Int → Int) → Value → Value → Thread Value
intOp op (IntVal x) (IntVal y) = return $ IntVal $ op x y
intOp _ _ _ = throw EvalError

intBoolOp :: (Int → Int → Bool) → Value → Value → Thread Value
intBoolOp op (IntVal x) (IntVal y) = return $ IntVal $
    if op x y then 1 else 0
intBoolOp _ _ _ = throw EvalError

boolOp :: (Bool → Bool → Bool) → Value → Value → Thread Value
boolOp op (IntVal x) (IntVal y) = return $ IntVal $
    if op (x > 0) (y > 0) then 1 else 0
boolOp _ _ _ = throw EvalError
```

```
lambdaGetVar :: Value → Thread Value
lambdaGetVar (Lambda x _) = return $ Variable x
lambdaGetVar _ = throw EvalError

lambdaGetVal :: Value → Thread Value
lambdaGetVal (Lambda _ y) = return y
lambdaGetVal _ = throw EvalError

lambdaSetVar :: Value → Value → Thread Value
lambdaSetVar (Lambda _ y) (Variable x) = return $ Lambda x y
lambdaSetVar _ _ = throw EvalError

lambdaSetVal :: Value → Value → Thread Value
lambdaSetVal (Lambda x _) y = return $ Lambda x y
lambdaSetVal _ _ = throw EvalError

funcCallGetFunc :: Value → Thread Value
funcCallGetFunc (FuncCall f _) = return f
funcCallGetFunc _ = throw EvalError

funcCallGetArg :: Value → Thread Value
funcCallGetArg (FuncCall _ a) = return a
funcCallGetArg _ = throw EvalError

funcCallSetFunc :: Value → Value → Thread Value
funcCallSetFunc (FuncCall _ a) f = return $ FuncCall f a
funcCallSetFunc _ _ = throw EvalError

funcCallSetArg :: Value → Value → Thread Value
funcCallSetArg (FuncCall f _) a = return $ FuncCall f a
funcCallSetArg _ _ = throw EvalError
```

5 Mutate.lhs

```
module Mutate (
    mutate,
    randomValue,
) where

import Value
import State
import Builtins

import System.Random
import Control.Monad.Random

type RandM = Rand StdGen

mutateP :: Double
mutateP = 0.01

mutateParP :: Double
mutateParP = 0.2

mutateFuncP :: Double
mutateFuncP = 0.3

mutateTypeP :: Double
mutateTypeP = 0.1

mutateInt :: Int → RandM Int
mutateInt x = do
    b ← getRandom
    return $ if b then x + 1 else x - 1

randInt :: RandM Int
randInt = getRandomR (-5, 5)

randIntVal :: RandM Value
randIntVal = randInt >>= return ∘ IntVal

randPrimFunc :: RandM Value
randPrimFunc = do
    i ← getRandomR (0, length primFuncs - 1)
    return $ primFuncs !! i

randLambda :: RandM Value
randLambda = do
    x ← randInt
    v ← randomValue
    return $ Lambda x v

randVariable :: RandM Value
randVariable = randInt >>= return ∘ Variable
```

```
randFuncCall :: RandM Value
randFuncCall = do
    f ← randomValue
    a ← randomValue
    return $ FuncCall f a

mutateInplace :: Value → RandM Value
mutateInplace (IntVal x) = mutateInt x >>= return ∘ IntVal
mutateInplace (PrimFunc _ _) = randPrimFunc
mutateInplace (Lambda x v) = do
    b ← getRandom
    if b < mutateParP then do
        x' ← mutateInt x
        return $ Lambda x' v
    else do
        v' ← mutateInplace v
        return $ Lambda x v'
mutateInplace (Variable x) = mutateInt x >>= return ∘ Variable
mutateInplace (FuncCall f a) = do
    b ← getRandom
    if b < mutateFuncP then do
        f' ← mutateInplace f
        return $ FuncCall f' a
    else do
        a' ← mutateInplace a
        return $ FuncCall f a'

randomValue :: RandM Value
randomValue = do
    b ← getRandomR (0, 4)
    case b :: Int of
        0 → randIntVal
        1 → randPrimFunc
        2 → randLambda
        3 → randVariable
        4 → randFuncCall

mutateValue :: Value → RandM Value
mutateValue x = do
    b ← getRandom
    if b < mutateTypeP then randomValue
    else mutateInplace x

mutate :: Thread ()
mutate = do
    b ← liftRandom getRandom
    when (b < mutateP) $ do
        n ← getSize
        i ← liftRandom $ getRandomR (0, n - 1)
        x ← getCell i
        x' ← liftRandom $ mutateValue x
        setCell i x'
```

6 Rep.lhs

```
module Rep (
    runStep,
    runN,
) where

import Value
import State
import Eval
import Mutate

import Control.Monad.Random

import qualified Data.Map as Map

randomThread :: Thread Value
randomThread = do
    n ← getSize
    i ← liftRandom $ getRandomR (0, n - 1)
    setCellPos i
    cell ← getCell i
    eval cell

runStep :: ([WorldState], [Thread Value]) → ([WorldState], [Thread Value])
runStep (states, threads) = (states'', threads'') where
    (threads', states') = unzip
        [runThread s (mutate >> t) | (s, t) ← zip states threads]
    restartThread (Left err) = randomThread
    restartThread (Right (Left t)) = t
    restartThread (Right (Right _)) = randomThread
    threads'' = map restartThread threads'
    univ = univMap $ head states
    univ' = Map.union (Map.unions $ map univEdits states') univ
    updateState state = state { univMap = univ', univEdits = Map.empty }
    states'' = map updateState states'

initialize :: Int → Int → Int → ([WorldState], [Thread Value])
initialize nCells nThreads seed = (states, threads) where
    rand = do
        cells ← sequence $ replicate nCells randomValue
        seeds ← sequence $ replicate nThreads getRandom
        return (cells, seeds)
    (cells, seeds) = evalRand rand $ mkStdGen seed
    univ = Map.fromList $ zip [0..] cells
    makeState s = WorldState { univMap = univ,
                               univSize = nCells,
                               univEdits = Map.empty,
                               envMap = Map.empty,
                               randomGen = mkStdGen s,
                               cellPos = 0 }
    states = [makeState s | s ← seeds]
    threads = replicate nThreads randomThread
```

```
runN :: Int → Int → Int → Int → [WorldState]
runN nCells nThreads seed n =
    fst $ iterate runStep (initialize nCells nThreads seed) !! n
```