Little Self-Replicating Programs

Alex Gajewski (apg2162)

December 18, 2019

1 Introduction

In this project, I built a simple open-ended ALife simulator. Open-ended is a keyword here. Most ALife simulators use an explicit evolutionary process, with organisms that have some genetic code and some reproduction mechanism. They also usually have some sense of evolutionary fitness that the organisms are trying to optimize.

In this project, I've done away all these things and simulated at a slightly lower level, where you just have little bits of code that can modify each other and themselves, and a bit of randomness thrown in. The idea is that something like reproduction would probably emerge, because if any such idea happened to emerge by chance, it would quickly expand and take over everything. The only question is how long it would take until such a thing emerged.

More concretely, the idea is this: there is a list of code expressions (imagine Lisp S-expressions for something specific) that represents the universe, and there is some fixed number of threads of execution, representing energy in this universe (this is where parallelization could give a speedup). Each thread of execution is always evaluating some expression in the list, and if it ever finishes, it jumps to another random one and starts up again.

There is also some base rate of mutation, by which randomly selected code expressions are mutated according to some mutation rule. Importantly, some of the instructions allow the code expressions to read and write their own and each others' code. This is done through relative addressing so that the same expression can be used for recursion anywhere in the list. For example, an expression should ask for (peek 0) to get itself, (peek 1) for its neighbor to the right, (peek -1) for its neighbor to the left, etc. This mechanism, together with the ability to pass execution to one of these other expressions (e.g. ((expr -1))) ought to be enough to make the whole thing Turing-complete.

At the beginning of the simulation, the universe list is filled with randomly generated code expressions, and the threads of execution are be assigned to randomly selected expressions. At this point, the game will be to see how long it takes for recursion to occur (all you need is one of the expressions to be ((expr 0)) to capture the execution thread indefinitely, until a mutation kills the program), and to try to study any other patterns that emerge.

2 Code

This section contains a walkthrough of all of the code in the simulator. If something doesn't make sense, feel free to email me at `apg2162@columbia.edu`.

2.1 Value.lhs

This module contains declarations of the basic types that we'll be using throughout the rest of the code. It also contains a few little helper functions that didn't have better homes. The following language extensions just make things a bit easier, letting us automatically derive a few typeclass instances and implement the `MonadState` typeclass.

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

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

We'll be using transformers to build up the `Thread` monad, so we need the following imports:

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

We also need the `Rand` monad to deal with mutations and random initialization. There is a `RandT` transformer, but since it's just a wrapper for `StateT` and since we're already using one of those to keep track of the `WorldState`, I thought it would be more straightforward to add a random generator to the `WorldState` and make a `liftRandom` helper function (see below).

```
import Control.Monad.Random
import System.Random
```

For parallelization, we're going make `NFData` instances for all of the relevant data structures, so that we can force deep enough parallel evaluation at each step of execution.

```
import Control.DeepSeq
import Control.Parallel.Strategies
```

Basically everything is currently implemented with maps, even the universe of cells, which would more reasonably have been implemented as an array. This was just for simplicity. `Array.Diff` is still experimental, and it would have been annoying to wrap everything in `ST` or `IO`, so I just went with maps for everything. Future versions could use more efficient data structures, but since the point of this project was (1) to be a proof of concept and (2) to try to get a parallelization speedup, it seemed fine to have the sequential code be a bit inefficient.

```
import qualified Data.Map as Map
```

The `Value` type represents both code and data in our small interpreted language. It only has support for integers and functions, and the functions are all fexprs for simplicity.

```
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 ++ ")"
```

The following is just a helpful type alias, because there are lots of maps from integers to values.

```
type ValueMap = Map.Map Int Value
```

The following is our error type, which causes the evaluation of a thread to halt prematurely. A different version of this code could have different `EvalError` constructors to allow for easier inspection of what the code is doing, but for this first version I went with a single constructor.

```
data EvalError = EvalError
```

The `WorldState` type contains all of the state data a single thread of execution needs in order to operate. The `univMap` is a read-only map from cell-number to value (which should be thought of as an array), and is the same across all threads. The `univSize` field is the size of the universe, i.e. the number of cells. The `univEdits` map contains the current thread's edits to the universe since the last time the different `univMap`s have been synchronized. When a cell's value is queried, it is first searched for in `univEdits`, and then in `univMap`. When a cell's value is written to, it is written in `univEdits`. The `envMap` contains the current thread's local scope. This is unique to each thread. This is mainly used for arguments and local variables. The `randomGen` field is the random generator for the current thread. The generators for different threads are initialized with different random seeds. Finally, the location of the cell the thread is currently evaluating is stored in `cellPos`.

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

A `Thread` is an identity coroutine (meaning it can be paused, but doesn't generate a value until it's finished) that can fail with an `EvalError`, and always has a `WorldState`, even if it has failed.

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

The following is a helper instance making it easier to access the internal `WorldState`.

```
instance MonadState WorldState Thread where
    get = Thread $ lift $ get
    put = Thread ∘ lift ∘ put
```

And as promised, here are the relevant `NFData` instances:

```
instance NFData (Thread a) where
    rnf t = seq t ()

instance NFData EvalError where
    rnf e = seq e ()

instance NFData Value where
    rnf (IntVal x) = seq x ()
    rnf (PrimFunc name f) = seq name $ seq f ()
    rnf (Lambda var val) = seq var $ deepseq val ()
    rnf (Variable var) = seq var ()
    rnf (FuncCall f a) = deepseq f $ deepseq a ()

instance NFData WorldState where
    rnf (WorldState { univMap,
                      univSize,
                      univEdits,
                      envMap,
                      randomGen,
                      cellPos,
                      evalTime }) = runEval $ do
        rdeepseq univMap
        rseq univSize
        rdeepseq univEdits
        rdeepseq envMap
        rseq randomGen
        rseq cellPos
        rseq evalTime
        return ()
```

We don't need a `MonadError` instance since we never need to catch any errors, so this is essentially just the `throwError` method from the `MonadError` typeclass.

```
throw :: EvalError → Thread a
throw = Thread ∘ lift ∘ throwError
```

As the name suggests, the `pause` function pauses the current thread.

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

The `runThread` function just runs the whole monad transformer and gets it into a form we can work with directly. This is done at every step of execution.

```
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)
```

The following is just a helper function that lifts an action from the `Rand` monad to the `Thread` monad.

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

2.2 State.lhs

This module contains some helper functions for dealing with the `WorldState`.

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

import Value

import Control.Monad.State

import System.Random

import qualified Data.Map as Map
```

The `getVar` function just gets a variable from the local execution scope, or throws an error if it's not found.

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

The `setVar` function just sets a variable in the local execution scope.

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

The `getCell` function just gets the value of a cell from the universe, looking first in the current thread's edits and then in the read-only `univMap`. It causes the program to crash if the requested cell is out of bounds.

```
getCell :: Int → Thread Value
getCell x = do
    state ← get
    return $ case univEdits state Map.!? x of
        Just y → y
        Nothing → univMap state Map.! x
```

The `setCell` function just sets the value of a cell in the thread's local `univEdits`.

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

The following just gets the index of the cell the current thread is evaluating. This is used for some of the locality-sensitive builtin functions.

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

The following just sets the index of the cell the current thread is evaluating.

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

The following just gets the size of the universe, i.e. the total number of cells.

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

The following just resets the `evalTime` to 0. It is called when a thread starts evaluating a new cell.

```
resetEvalTime :: Thread ()
resetEvalTime = do
    state ← get
    put $ state { evalTime = 0 }
```

2.3 Eval.lhs

This module is very short, and only exists because it can't go anywhere else. It only contains one function, the `eval` function, which evaluates a value in the current thread.

```
module Eval (
    eval,
) where

import Value
import State

eval :: Value → Thread Value
```

Values are all autoquoted, as in Lisp:

```
eval x@(IntVal _) = return x
eval x@(PrimFunc _ _) = return x
eval x@(Lambda _ _) = return x
```

Evaluating a variable just gets it from the local environment:

```
eval (Variable x) = getVar x
```

Evaluating a function is the only time the current thread gets paused, and it gets paused between when the result is evaluated and when it is returned. Attempting to evaluate something that isn't a function kills the thread.

```
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
```

2.4 Builtins.lhs

This module contains all the builtin functions that can be used. Hopefully I didn't forget anything that prevents the interpreter from being Turing-complete.

```
module Builtins (
    primFuncs,
) where

import Value
import State
import Eval
```

There's just one export, the `primFuncs` export, which is just a list of `PrimFunc`s.

```
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]
```

The following are just some helper functions that make defining multi-parameter functions and macros easier. The difference is that functions automatically evaluate their parameters, but macros do not. Fundamentally they're both fexprs, the only reason there are both is to reduce repetition in function definitions. There's probably some crazy dependent-type way to make these helpers work for functions of any arity, but since there are only two of each type, it seemed fine to do it by hand.

```
func1 :: String -> (Value -> Thread Value) -> Value
func1 name f = PrimFunc name $ \x -> 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
```

The following is just an if macro. There aren't booleans in the language, so positive integers are treated as true and negative ones as false.

```
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
```

The following is a macro that sets a local variable (recall that these are unique to each thread of execution).

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

The peek and poke functions read from and write to cells, respectively. The names are a reference to early BASIC machines.

```
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
```

The `intOp`, `intBoolOp`, and `boolOp` functions are helpers for defining builtin binary operators on integers and "bools", which are also just integers. The operators that output booleans output 1 for true and 0 for false.

```
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
```

The next few functions are for metaprogramming, allowing the construction and deconstruction of lambdas and function calls. They should be helpful in allowing actual self-replication.

```
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
```

2.5 Mutate.lhs

This module contains actions that mutate contents of cells and generate new random values. The actions are all in the `Rand` monad instead of in the `Thread` monad because they only need the random generator, not any other part of the `WorldState`, so it would have been overkill to give them an entire `Thread` coroutine. Also because they're not just used from within the `Thread` monad, but are also used for initializing the universe at the beginning of execution.

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

import Value
import State
import Builtins

import System.Random
import Control.Monad.Random
```

The following is just a helper type alias for the monad we're going to be doing everything in.

```
type RandM = Rand StdGen
```

The following represents the probability that any mutation is going to occur this step.

```
mutateP :: Double
mutateP = 0.01
```

The following represents the probability that the parameter of a `Lambda`, as opposed to its body, will be mutated.

```
mutateParP :: Double
mutateParP = 0.2
```

The following is the probability that a function, as opposed to its argument, will be mutated in a function call.

```
mutateFuncP :: Double
mutateFuncP = 0.3
```

The following is the probability that an entirely new random value will be generated, as opposed to the differential modifications that are otherwise performed.

```
mutateTypeP :: Double
mutateTypeP = 0.1
```

When an int is mutated, it is randomly incremented or decremented with equal probability.

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

When a new integer is generated, it is selected from the range $[-5, 5]$.

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

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

When a new primitive function is generated, it is selected at random from the list of primitive functions.

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

When a new lambda is generated, its parameter is a random integer from the range $[-5, 5]$, and its body is a randomly generated value.

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

When a variable is generated, it is selected at random from the range $[-5, 5]$.

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

When a function call is generated, both the function and the argument are randomly generated values. If the function is an integer this will result in the thread crashing, but that's sufficiently low probability that it wouldn't have been worth making a separate random function generator.

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

The `mutateInplace` function just puts together all of the above probabilities and mutators and applies them to an arbitrary value.

```
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'
```

When a new random value is required, its type is selected at random from the 5 different possible types (ints, primitive functions, lambdas, variables, and function calls), with equal probability. A different version of the code could have different probabilities for each of the types, but there are already a lot of parameters to deal with, and it's not clear how these probabilities should deviate from uniform for better performance.

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

The `mutateValue` function just generates a new random value with probability `mutateTypeP`, and mutates the existing value otherwise.

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

When we mutate a thread, we first check whether any mutations are going to occur (probability `mutateP`), and if so, we pick a random cell to mutate, and then mutate it.

```
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'
```

2.6 Rep.lhs

This module contains the heart and soul of the simulator, the code that sets up the initial state, and the code that takes a state and runs one step of simulation on it. That is, this module contains the initial value and the update rule of the dynamical system we're building.

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

import Value
import State
import Eval
import Mutate

import Control.Monad.Random

import Control.Parallel.Strategies

import qualified Data.Map as Map
```

The `randomThread` function assigns a thread to a random cell to evaluate, and starts it evaluating that random cell.

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

To run a step of simulation, we tell the threads to mutate the universe, and then we run the threads for one step of execution, randomly restart any threads that have finished evaluating their assigned cells, then merge edits to the universe and write the results back to the thread states. In terms of parallelization, it is relatively straightforward, just doing a parallel map over the `runThreads`.

```
runStep :: ([WorldState], [Thread Value]) → ([WorldState], [Thread Value])
runStep (states, threads) = (states'', threads'') where
    evalList = [runThread s (mutate >> t) | (s, t) ← zip states threads]
    (threads', states') = unzip (evalList `using` parList rdeepseq)
    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,
                                evalTime = evalTime state + 1 }
    states'' = map updateState states'
```

At the beginning of the simulation, we set each cell to a random value, generate some random seeds to give each thread a different random generator, and start each thread on evaluating a random cell.

```
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,
                               evalTime = 0 }
    states = [makeState s | s ← seeds]
    threads = replicate nThreads randomThread
```

The following is just a helper function that will run the simulation for n steps, and output the longest evaluation times for each thread.

```
runN :: Int → Int → Int → Int → [Int]
runN nCells nThreads seed n =
    runN' (initialize nCells nThreads seed) (replicate nThreads 0) n where
        runN' state maxs 0 = maxs
        runN' state maxs m = runN' (states, threads) maxs' (m - 1) where
            (states, threads) = runStep state
            maxs' = [max (evalTime s) m | (s, m) ← zip states maxs]
```

3 Results and Discussion

Empirically, parallelization helps a decent amount, though definitely far from linear. A simple experiment with 100 cells, 100 threads of execution, and running for 10000 steps took 5.49 seconds to run in a single Haskell Execution Context (HEC), 5.16 seconds to run on two HECs, and 2.96 seconds to run on three HECs. Studying the eventlog, it seems that the main difficulty is that each step of work is so small (basically just evaluating a single function call with depth 1) that the overhead of sending the work to an HEC is comparable to the actual work of evaluating the expression.

One way of making the algorithm more parallelizable in the future could be to pause execution more infrequently, for example by pausing with stochastically with some fixed probability, say 10%. Then each thunk would give the HECs more work to do. On the other hand, this would cause some thunks to take much longer to evaluate than others, and since the different steps of simulation are necessarily synchronized in lock-step, this could leave some HECs starved, unless there are many more threads of execution than HECs.

As for whether or not self-replication emerged, unfortunately it seems it did not in this first version. In fact, the longest expression took just 7 steps to evaluate, indicating that the simulation did not even discover recursion. The probable cause for this is that there were too many builtin functions; future versions of the simulator could try to simplify the language used in the interpreter to make recursion easier to discover. Other improvements could be to use `Array.Diff` instead of maps for the universe, or perhaps the `ST` monad, to speed up the number of steps per second that the simulator is able to run, enabling the simulator to run for longer and allowing more complex behaviors to emerge.