Genetic Space Invaders

Programación Funcional

Instituto Tecnológico de Buenos Aires

Buenos Aires, Argentina

2018

Integrante: Juan Manuel Alonso - jalonso@itba.edu.ar - 56080

Profesores:

- Martinez Lopez, Pablo Ernesto *Fidel*
- Pennella, Valeria Verónica

Índice

Introducción

El objetivo de este trabajo es implementar *Space Invaders,* un videojuego de arcade diseñado en Japón y lanzado en 1978 cuyo objetivo es eliminar oleadas de alienígenas con un cañón láser y obtener la mayor cantidad de puntos posible. En este sentido, se agregó una implementación de un algoritmo genético que consta de reproducir miembros de una población en base a cambios en sus genes.

Se optó desarrollar el juego en el lenguaje Elm, un lenguaje funcional apto para crear interfaces gráficas basadas en *web browsers*. Posee un conjunto restringido pero poderoso de construcciones de lenguaje, incluyendo las expresiones tradicionales *if, let* para estados locales y *case*. Al ser funcional, soporta funciones anónimas, funciones como argumentos y aplicación parcial por default. Su semántica incluye valores inmutables, funciones sin *side effects*, tipado estático con inferencia de tipos y una estructura de datos *register*. Los programas en Elm renderizan HTML a través de un DOM virtual y utilizan JavaScript.

Model - Update - View

La arquitectura del lenguaje ofrece el clásico estructuramiento de un juego en ***Model*** (modelo), estado de los actores del juego, ***Update*** (actualizar), cómo se actualizan esos estados, y ***View*** (vista), cómo se reflejan frente al usuario esos cambios. Además, el lenguaje presentó una estrategia de programación reactiva en base a un manejo de mensajes, comandos y suscripciones a eventos.

Model

Los modelos más relevantes son los siguientes.

```
type State = Play | Pause | Start | Over

type alias Spaceship =
   { x : Float
   , y : Float
   , vx : Float
   , vy : Float
   }
```

```
type alias Bullet =
    { x : Float
    , y : Float
    , vx : Float
    , vy : Float
    , hit : Bool
    }

type alias Invader =
    { x : Float
    , y : Float
    , vx : Float
    , vy : Float
    , xProbChange : Float
    , yProbChange : Float
    , seedX : Seed
    , seedY : Seed
    , wasHit : Bool
    }

type alias Game =
    { keysDown : Set KeyCode
    , windowDimensions : ( Int, Int )
    , state : State
    , spaceship : Spaceship
    , invaders : List Invader
    , bullets : List Bullet
    , bestSolution : ( IntermediateValue Dna, Seed )
    , currentTime : Time
    , hasSpawned : Bool
    , score : Int
    }
```

Notar que Invader cuenta con una probabilidad de cambio en cada dirección bidimensional, y sujeta a cada una de ella, una denominada *semilla* necesaria para producir valores aleatorios. En cuanto a Game, más allá de parámetros generales, se destaca la mejor solución hasta el momento, descripta por un valor intermedio de adn (un conjunto de genes de los parámetros de Invader) y una semilla cambiante que garantiza aleatoriedad en llamados sucesivos.

Update

En relación a la parte de actualización del modelo, se destacan las siguientes funciones:

```
updateInvaders : Time -> List Invader -> List Bullet -> List Invader
updateInvaders t invaders bullets =
    List.filter (\i -> not i.wasHit) (List.map (\i -> updateInvader t bullets i) invaders)


updateInvader : Time -> List Bullet -> Invader -> Invader
updateInvader t bullets invader =
    if List.any (\b -> within b invader) bullets then
        { invader | wasHit = True }
    else
        decideMovement t invader


updateBullets : Time -> List Bullet -> List Invader -> List Bullet
updateBullets t bullets invaders =
    List.filter (\b -> not b.hit) (List.map (\b -> updateBullet t invaders b) bullets)


updateBullet : Time -> List Invader -> Bullet -> Bullet
updateBullet t invaders bullet =
    if (bullet.y >= (halfHeight - 40)) || List.any (\i -> within bullet i) invaders then
        { bullet | hit = True }
    else
        physicsUpdate t bullet
```

Figura 1. Captura de pantalla. Actualización de invasores y balas.

Notar el poder de las funciones de alto orden en ambas actualizaciones. En una línea, primero se actualizan los Invaders con *map* marcandolos si fueron abatidos por alguna bala, Bullet, (de lo contrario, se decide un movimiento aleatorio) y segundo se filtran con *filter* aquellos marcados anteriormente. Para el caso de la actualización de las Bullets es casi idéntico. Se decidió no modularizar estas funciones frente a la opción de conservar la legibilidad general de ambas.

```
calculateFitness : Dna -> List Invader -> Float
calculateFitness dna invaders =
    toFloat
        (List.length
            (List.filter
                (\invader ->
                    (invader.xProbChange == dna.genes.xProbChange)
                        || (invader.yProbChange == dna.genes.yProbChange)
                        || (invader.vx == dna.genes.vx)
                        || (invader.vy == dna.genes.vy)
                )
                invaders
            )
        )


updateSolution : Float -> ( IntermediateValue Dna, Seed ) -> ( IntermediateValue Dna, Seed )
updateSolution newFitness ( IntermediateValue p pd ng, seed ) =
    ( IntermediateValue p { dna = { genes = pd.dna.genes, fitness = newFitness }, points = newFitness } ng, seed )
```

Figura 2. Captura de pantalla. Cálculo de aptitud y actualización de solución.

En cuanto al algoritmo genético, se destaca el cálculo de la aptitud o *fitness* de los Invaders para que el primero logre puntuar las soluciones intermedias que vaya proponiendo. Esto se hace contando la cantidad de Invaders por cada *frame* del juego que comparten algún atributo con el adn ideal hasta el momento debido a que el algoritmo puede mutar algún gen en su paso evolutivo, que sería imposible o engorroso de registrar. Cuanto más persista un Invader en el juego sin que el usuario lo mate, mayor valor cobran sus genes.

Con el resultado de *calculateFitness*, la segunda función incluida actualiza la mejor solución hasta el momento para luego reinsertarla en el algoritmo y que este la utilice en su próximo paso evolutivo. De esta manera, se procede a crear tres nuevos alienígenas cada dos segundos con el adn ideal resultante de dicho paso, haciendo uso de semillas para aleatoriedad y de los genes de la solución óptima hasta el momento.

View

Si bien la vista no fue un tema trascendental en cuanto a lo funcional, es notable destacar el poder expresivo del lenguaje en cuanto al manejo de estados y cómo se deben ver reflejados cambios específicos. Es el caso de *messageStatus*, que según el estado del juego, cambia de estilo y de texto de mensaje para devolver un elemento editable en HTML.

```
messageStatus : State -> Element
messageStatus state =
    case state of
        Play ->
            txt identity ""

        Pause ->
            txt (Text.italic >> Text.color yellow) pauseMessage

        Start ->
            txt (Text.italic >> Text.height 12 >> Text.color yellow) startMessage

        Over ->
            txt (Text.italic >> Text.bold >> Text.color red >> Text.height 18) overMessage
```

Figura 3. Captura de pantalla. Mensaje del estado del juego.

Algoritmo genético

El algoritmo genético, provisto por Charlie K. (https://github.com/ckoster22) en su paquete Elm Genetic y correspondiente módulo, ha sido una parte fundamental en el desarrollo del juego y de su aplicación. El objetivo fue independizar el modelo del juego de los requerimientos del algoritmo. En este sentido se creó un módulo separado que funciona como intermediario entre Genetic y el juego en sí.

Como requerimiento del algoritmo, se definió el tipo Dna. Para afianzar la claridad del código, también se definió el tipo Genes, que convenientemente comparte propiedades con Invader.

```
type alias Genes =
    { vx : Float
    , vy : Float
    , xProbChange : Float
    , yProbChange : Float
    }
type alias Dna =
    { genes :
        Genes
    , fitness : Float
    }
```

Gracias a su practicidad, el algoritmo genético únicamente requiere para su su desenvolvimiento de un registro con funciones auxiliares que permitan.

```
myOptions : Options Dna
myOptions =
    { randomDnaGenerator = randDnaGenerator
    , evaluateSolution = evaluateSolution
    , crossoverDnas = crossoverDnas
    , mutateDna = mutateDna
    , isDoneEvolving = isDoneEvolving
    , method = MaximizeScore
    }
```

- *randDnaGenerator* : un generador aleatorio de Dna, necesario para la generación de nuevos alienígenas.
- *evaluateSolution*: función que contabiliza la aptitud de los Invaders. En el caso del trabajo es *fitness* explicada anteriormente.
- *crossoverDna*: como todo algorítmo genético, cuenta con una función para el cruzamiento y mutación de la población. *crossoverDna* toma las componentes de Dna en *x* de un *padre* y las de *y* del otro para formar un *hijo,* cuya *fitness* queda como el promedio.
- *mutateDna*: función que elegirá un gen aleatorio para mutar entre los del Dna provisto, y así producir un miembro mutado.
- *isDoneEvolving*: función que determina el corte del algoritmo (3000 iteraciones en total o si la aptitud llega al valor máximo de punto flotante, lo que ocurra primero).
- *method*: en vez de minimizar la penalidad de las soluciones, opción que también provee el algoritmo, se optó por maximizar su valor.

Condición de derrota

Al igual que otros juegos en tendencia, *Genetic Space Invaders* no tiene condición de victoria. El jugador puede seguir sumando puntaje (1 punto por cada alien) acribillando invasores (*Java Dukes* que atentan contra el paradigma funcional) con la *Lambda Shuttle*, al menos que los alienígenas se reproduzcan lo suficiente hasta llegar a 100 en cantidad.

Figura 4. Captura de pantalla. Mensaje de inicio con instrucciones.

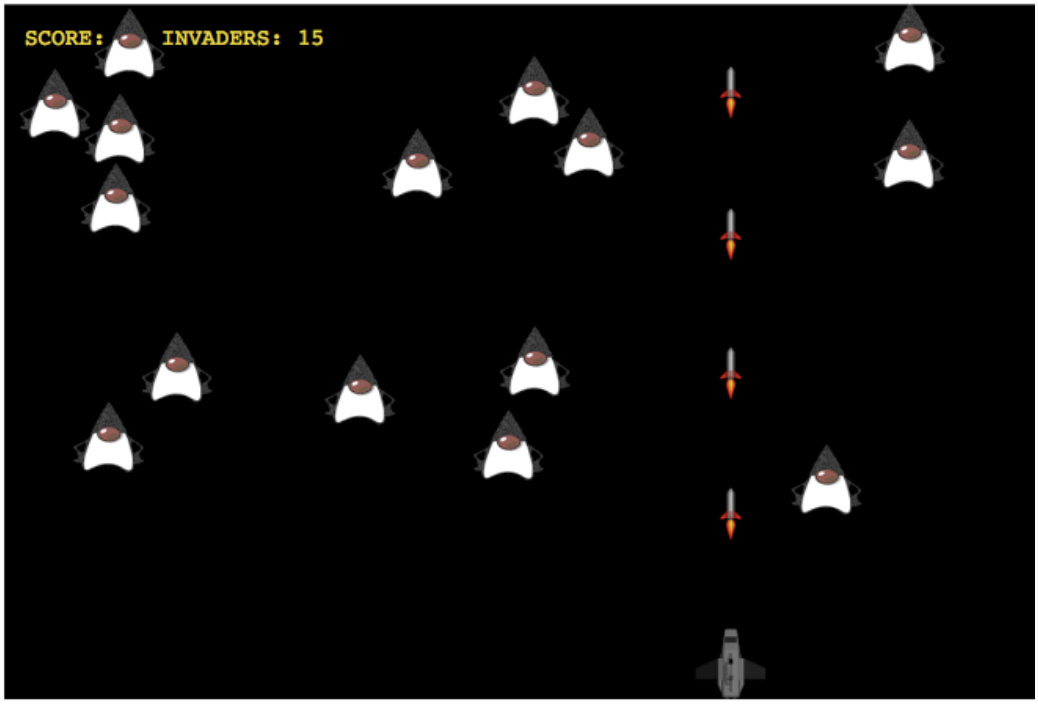

Figura 5. Captura de pantalla. Juego en curso.

Figura 6. Captura de pantalla. Juego perdido.

Extensiones

A futuro algunas mejoras son factibles:

- introducción de condición de victoria
- introducción de diferentes niveles de dificultad
- manipulación del usuario del algoritmo genético
- almacenar puntajes
- explorar otros métodos que precisen adecuadamente funciones de alto orden como *fold*

Instalación

Para instalar el juego de manera local desde el código en una computadora es necesario que cuente con macOS o Linux y contar con Elm ya instalado. Si no es así puede obtenerlo aquí: https://guide.elm-lang.org/install.html.

1. Dirigirse a una nueva ventana de la línea de comando/terminal.
2. Insertar los siguientes comandos.

   ```
   git clone https://github.com/j1nma/genetic-space-invaders.git
   cd genetic-space-invaders
   make
   make open
   ```
3. En caso de que `make open` no abra un navegador con el juego, ir a la carpeta genetic-space-invaders y abrir el archivo *index.html* que acaba de ser construido.

Pueden obviarse los pasos anteriores si se desea jugar directamente desde la página provista por el servicio que aloja este proyecto *online:* https://j1nma.github.io/genetic-space-invaders/.
El proyecto se encuentra alojado en https://github.com/j1nma/genetic-space-invaders.

Conclusión

Elm cuenta con algunas limitaciones en comparación a lenguajes funcionales más puros como Haskell. Por un lado, soporta *pattern matching* a medias, pues no permite más de una línea como definición de funciones. Esto implica que el *pattern matching* que suele encontrarse del lado de los parámetros se vea traducido es una expresión *case.* Por otro lado, no cuenta con funciones de alto orden genéricas, como lo son *map, filter, fold* y *apply*. En este caso cada función es expuesta por algún módulo que la utilice, por ejemplo, List.map.

En sintonía con lo anterior, y tal como lo aclara el fundador de Elm[1], es limitado el desarrollo de mónadas en Elm aunque sí se encuentra en módulos usados con frecuencia como Random o State. En un comienzo se logró crear para este trabajo un ejemplo de una mónada de estado definida detalladamente en Haskell [ver Figura 7], pero la propia naturaleza de Elm chocaba con esta intención a tal punto de hacer imposible la compatibilidad entre ambos.

También ocurrieron inconvenientes en cuanto a la frecuencia de refresco de los cuadros del juego, que presentaba problemas a la hora de actualizar las vistas de los modelos. Se terminó duplicando la misma, una decisión más empírica que respaldada por documentación sobre el tema, que no es demasiado explicativa.

En relación al algoritmo genético, también fueron necesarias algunas modificaciones a la implementación del paquete usado en relación a la exposición del tipo *IntermediateValue*, reduciendo la robustez general.

En conclusión, si bien se presentaron dificultades externas al paradigma funcional, el desarrollo del juego en Elm en conjunto a la aplicación de un algoritmo genético resultó fructífero, pues se aprovechó el poder expresivo y funcional del lenguaje y se puso a práctica lo visto en clase.

[1] Monads are also called 'computation builders': They allow for an elegant way of chaining computations with andThen. ***Elm wants to be a simple, easy to learn language, and therefore monads aren't really talked about (yet)***. I've tried to limit the jargon in the documentation to a minimum.
http://package.elm-lang.org/packages/folkertdev/elm-state/latest/State

```haskell
data ST s a = ST (s -> Maybe (s, a))

unST (ST f) = f

instance Functor (ST s) where
  fmap f (ST s) = ST (\sd -> case unST (ST s) sd of
                               Nothing -> Nothing
                               Just (ss, x) -> Just (ss, f x))

instance Applicative (ST s) where
  pure a = ST (\s -> Just (s, a))
  ST f <*> ST x = ST (\s -> case unST (ST f) s of
                               Nothing -> Nothing
                               Just (ss, g) -> unST (fmap g (ST x)) ss)

instance Monad (ST s) where
   return x = ST (\s -> Just (s, x))
   m >>= k = ST (\s -> case (unST m s) of
                          Nothing -> Nothing
                          Just (s', v) -> unST (k v) s')
   fail _ = ST (\s -> Nothing)

getState :: ST s s
getState = ST (\s -> Just (s,s))

setState :: s -> ST s ()
setState sn = ST (\s -> Just (sn, ()))

runST :: ST s a -> s -> Maybe a
runST m s = fmap snd (unST m s)

numsToStrings n | n `mod` 2 == 0 = "Even was here"
                | otherwise = "Odd was here"

stringify nv = fmap numsToStrings (return nv)

testState = ST (\s -> Just (s, 0))

testBind = unST((>>=) testState stringify) "A new state"

main = do line <- getLine
          let line' = reverse line
          putStrLn $ "You said " ++ line' ++ " backwards!"
          putStrLn $ "Yes, you really said" ++ line' ++ " backwards!"
```

Figura 7. Captura de pantalla. Intento de definición de mónada de estado.