Genetic Space Invaders

Programación Funcional

Trabajo Práctico Final

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. La motivación del trabajo surge del deseo de aplicar los conceptos vistos en clase del paradigma funcional mediante el entretenido desarrollo de un videojuego para navegadores. En este sentido, se decidió enriquecer el mismo con la implementación de un algoritmo genético que permite 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.

Se supuso que surgirían dificultades en la primer aproximación a Elm, lenguaje con el que el autor de este trabajo no había experimentado antes, en cuanto al aprendizaje de su sintaxis (no demasiado lejana a la de Haskell, vista en clase) y del flujo de trabajo con respecto al manejo de mensajes, comandos y suscripciones que el mismo utiliza. También se esperaron inconvenientes a la hora de implementar el algoritmo genético, comprender si este se adecuaba al problema del trabajo y analizar si los resultados del mismo coincidían con la dinámica del juego.

El informe presente consta del planteo del problema del trabajo, una sección técnica donde se detalla la arquitectura del sistema y decisiones técnicas y por último conclusiones donde se comenta sobre las barreras que se presentaron, lo aprendido en el trabajo y apreciaciones sobre el mismo.

Planteo del problema

Se dispuso a construir una única aplicación de página web, o SPA por sus siglas en inglés. Se planteó resolver los componentes necesarios por la arquitectura de Elm (definidos en el siguiente punto), resolver la integración del algoritmo genético al juego y definir cómo sería la dinámica del juego para el usuario.

Con respecto a la integración del algoritmo genético, primero se decidió investigar sobre el mismo y decidir si su uso era adecuado para el juego. Un algoritmo genético busca una solución lo suficientemente buena para un problema de optimización. Por lo general, un algoritmo genético requiere de la siguiente información: cómo generar una solución aleatoria, cómo evaluar una solución, cómo engendrar dos soluciones (*crossover)*, cómo modificar o *mutar* alguna solución y calcular si la actual solución es lo suficientemente buena.

Se eligió un paquete[1] en Elm que ofrecía esta funcionalidad. Se puede optar por una ejecución del algoritmo bloqueante, es decir, que busca una solución indefinidamente hasta encontrar una lo suficientemente buena, o de a pasos o *steps*, que cuenta con un paso inicial (*executeInitialStep)* y con un paso general (*executeStep),* la cual se eligió para el juego.

Se decidió que esta implementación era apta para el juego, pues el adn que utiliza para modelar los miembros de una población era compatible si se lo definía como un conjunto de los genes de los invasores: velocidad en *x*, velocidad en *y*, probabilidad de cambio de dirección en *x* y lo análogo en *y*.

Se dió paso a resolver las funciones auxiliares que representan la información requerida por el algoritmo, siendo las más importantes *crossover, mutateDna* y *evaluateSolution.* La función *crossover* opta por conservar los genes en *x* de un antecesor y los de *y* del otro, mientras que la función *mutateDna* elige un gen al azar para mutar y lo cambia por un valor aleatorio dependiendo del rango del mismo (velocidades de -100 a 100 ó probabilidades de 0 a 1). Por último, la función *evaluateSolution* calcula la aptitud o *fitness* de los *invaders* cómo la cantidad total de ellos en un *frame* que sigan vivos y compartan algún gen con la solución inicial. Para definir si una solución es lo suficientemente buena se determinó que *fitness* pueda ser el mayor valor de punto flotante.

En cuanto a la dinámica del juego para el usuario, la misma se planteó simple, contando con una página web única que conste de una pantalla de juego. Este pantalla puede alternar entre la del comienzo (ver Figura 1 del apéndice**)** y una pantalla de juego en curso (ver Figura 2). Las instrucciones se encuentran en la primera. 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) al *score* 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, cuando se pierde el juego (ver Figura 3).

[1] https://github.com/ckoster22

Sección técnica

Cuando se construye una aplicación *frontend* en Elm, se utiliza un patrón denominado Elm *architecture*. Este patrón provee una manera de crear componentes reutilizables, combinables y compuestas en sintonía con una estrategia de programación reactiva. Esta arquitectura cuenta con componentes que debieron ser bien definidas para proceder con el juego.

En primer lugar, los *mensajes (messages)*: es todo aquello que ocurra en la aplicación que las componentes puedan reaccionar. En la aplicación de *Genetic Space Invaders*, se definieron los mensajes KeyDown, KeyUp, WindowResize, Tick, NoOp y OnTime, los primeros relevantes a los visual y los últimos al manejo del tiempo.

En segundo lugar, la *vista (view)*: una función que recibe un modelo (en este caso, el juego o *Game*) y retorna un mensaje de tipo Html Msg que será renderizado para que el usuario observe algo en el navegador.

En tercer lugar, la *actualización (update)*: una función que será llamada cada vez que se reciba un mensaje. Responde a los mensajes definidos anteriormente que actualicen el modelo y retornen comandos como sea necesario.

En cuarto lugar, las *suscripciones (suscriptions)*: se utilizan para *escuchar* datos externos a la aplicación. En el caso del trabajo se encuentran la suscripción a evento del teclado KeyDown y KeyUp, cambios del tamaño de la ventana y el *tick* para refrescar los cuadros de animación.

En quinto lugar, los comandos *(commands):* es la manera en Elm de decir en tiempo de ejecución qué ejecutar que involucre *side effects*. Para el juego son *initialSizeCmd* (determina el tamaño de la ventana del juego), *getTime* (obtiene la hora actual) y *none* (comando nulo, no produce efectos).

Por último, *Html.program* hace un compendio de todo lo mencionado y devuelve un elemento Html que puede ser renderizado en una página web. Cabe destacar que, como en todo lenguaje funcional, cada función retorna un valor. *Side effects* de las funciones en el sentido tradicional están prohibidas por el diseño del lenguaje; es por esto que Elm toma una apreciación alternativa a la hora de modelarlos. En esencia, una función retorna un valor de comando con el nombre del efecto deseado. Debido a la arquitectura Elm, el *main Html.program* es el recipiente de este valor de comando y *update* contiene la lógica para ejecutar el comando nombrado.

Model - Update - View

La arquitectura del lenguaje ofrece el clásico estructuramiento de un juego en módulos: ***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.

Model

Los modelos más relevantes son los siguientes.

```
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
    }


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

El juego, *Game*, está descripto por un valor intermedio de adn (un conjunto de genes de los parámetros de Invader), una semilla cambiante que garantiza aleatoriedad en llamados sucesivos, teclas presionadas, dimensión de la ventana de juego, la nave espacial, una lista de los invasores, una lista de los misiles o balas que lanzó la nave, el estado del juego, la hora, un *booleano* para saber si ya se generaron nuevos invasores cada 2 segundos y el puntaje del jugador.

Los estados del juego son *Play*, cuando el jugador está jugando, *Pause*, cuando el jugador oprima 'p' para pausar el juego, *Start,* cuando el jugador oprima 's' para comenzar o reanudar el juego y *Over,* cuando los *invaders* llegan a 100 y el jugador pierde el juego. Estos estados determina cambios en *Game*, que serán llevados a cabo por el módulo de *Update* y reflejados por el módulo *View.*

```
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
    }
```

La nave espacial está modelada como un registro con coordenadas y velocidades bidimensionales que serán alteradas por las flechas en el teclado que oprima el jugador. Así es que con estos parámetros el jugador observa el movimiento físico de la naves, invasores y misiles. Los misiles de la nave están modelados por *Bullet,* que también posee coordenadas y velocidades bidimensionales y un *booleano* que representa si ese misil llegó a un invasor o no. Por último, los *Invaders,* modelados con posición y velocidades en *x* y en *y,* probabilidades de cambio en dichas direcciones, semillas también en ambas direcciones y un *boolean* que indica si fue abatido por un misil. Las probabilidades de

cambio son utilizadas en el momento de decidir el movimiento de un invasor. Este puede colisionar contra algún borde del juego, lo que implica cambiar el sentido de dirección de su movimiento, o de lo contrario, producir un movimiento aleatorio, que implica analizar estas probabilidades para darle a la dinámica del juego una sensación menos predecible. Las semillas o *seeds,* que se actualizan en cada actualización de un invasor, son necesarias para obtener valores aleatorios, ya que de lo contrario el comportamiento del invasor (o de cualquier entidad que utilice aleatoriedad) sería siempre constante o reproducible e iría en detrimento de la dinámica del juego.

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
```

Aquí se puede notar el poder de las funciones de alto orden en ambas actualizaciones. En una línea, primero se actualizan los Invaders con *map* marcándolos si fueron abatidos por alguna misil (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 )
```

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
```

Algoritmo genético

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


type alias Dna =
    { genes :
        Genes
    , fitness : Float
    }
```

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*. Gracias a su practicidad, el algoritmo genético únicamente requiere para su desenvolvimiento un registro con funciones auxiliares, introducidas en la sección técnica. Estas funciones son combinadas como *Options* y son entregadas al algoritmo en cada paso evolutivo.

Para verificar que el algoritmo genético produce resultados esperados que justifiquen su implementación y le den personalidad al juego, se utilizó una herramienta del navegador *Safari* y otra de *debugging* de Elm para observar en tiempo real qué valores maneja. De esta manera, la Figura 4 del apéndice ilustra las mejores soluciones devueltas por el algoritmo hasta el momento, comenzando desde arriba y continuando en el tiempo hacia abajo. Algunas líneas de esta figura contiene un número con fondo celeste en un lado izquierda que indica la cantidad de veces que la solución de esa línea fue tomada para reproducir (depende del *refresh rate* del navgeador). En síntesis, las líneas sin dicho número representan el resultado de un nuevo paso evolutivo, por eso su *fitness* es cero, y las líneas que lo poseen representan los ciclos de *update* donde no se ejecutó el paso evolutivo, por eso su *fitness* es 3, la cantidad de *invaders* nuevos que se engendran en cada paso. Esta decisión se basa en que no es práctico estar corriendo el algoritmo en cada cuadro del juego y, además, de que no es eficiente y necesario para el juego.

Para observar que el algoritmo produce cambios, se remarcaron en la Figura 4 tres cuadros rojos ejemplificando la evolución. En el cuadro 1, se observa que el gen de la velocidad en *x* es el mejor hasta el momento, pues en repetidos pasos sucesivos del algoritmo se conserva el valor. Más adelante, se puede ver que ese valor ya no permanece, pues a criterio del algoritmo, este fue mutado o reemplazado por otro de otro invasor en el *crossover*. En el cuadro 2, sucede lo mismo para la probabilidad de cambio en la dirección *x* por algunos momentos.. En el cuadro 3, el más revelador, se observa que la probabilidad de cambio en la dirección *y* se conserva también por sucesivos pasos evolutivos. De hecho, si además se observa el valor de *fitness* para esta solución, se nota que *fitness* está incrementando. Esto refleja que el algoritmo evalúa con mayor valor a esa solución y por ende se engendran *invaders* con ese adn. Sobre el final de este cuadro, se puede ver que el valor de *fitness* fue decrementando, pues el jugador estuvo matando invasores, lo que impacta en el cálculo de la aptitud de aquellos invasores y por ende en la solución que contiene sus genes.

Esto se tomó en repetidas ocasiones como evidencia de que la implementación del algoritmo genético es válida para el juego.

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, las funciones de alto orden genéricas, como lo son *map, filter, fold* y *apply*, no puede utilizarse sin importar explícitamente un módulo que lo exponga, por ejemplo, List.map. Esto no es así en Haskell, donde el módulo que las contiene está importado por defecto.

En sintonía con lo anterior, y tal como lo aclara el diseñador de Elm[2], 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, pero la propia sintaxis y arquitectura de Elm chocaba con esta intención a tal punto de hacer imposible la compatibilidad entre el lenguaje y la mónada de estado definida.

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, funcional y de programación reactiva del lenguaje y se puso en práctica lo visto en clase.

[2] 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

Apéndice

Figura 1. Captura de pantalla. Mensaje de inicio con instrucciones.

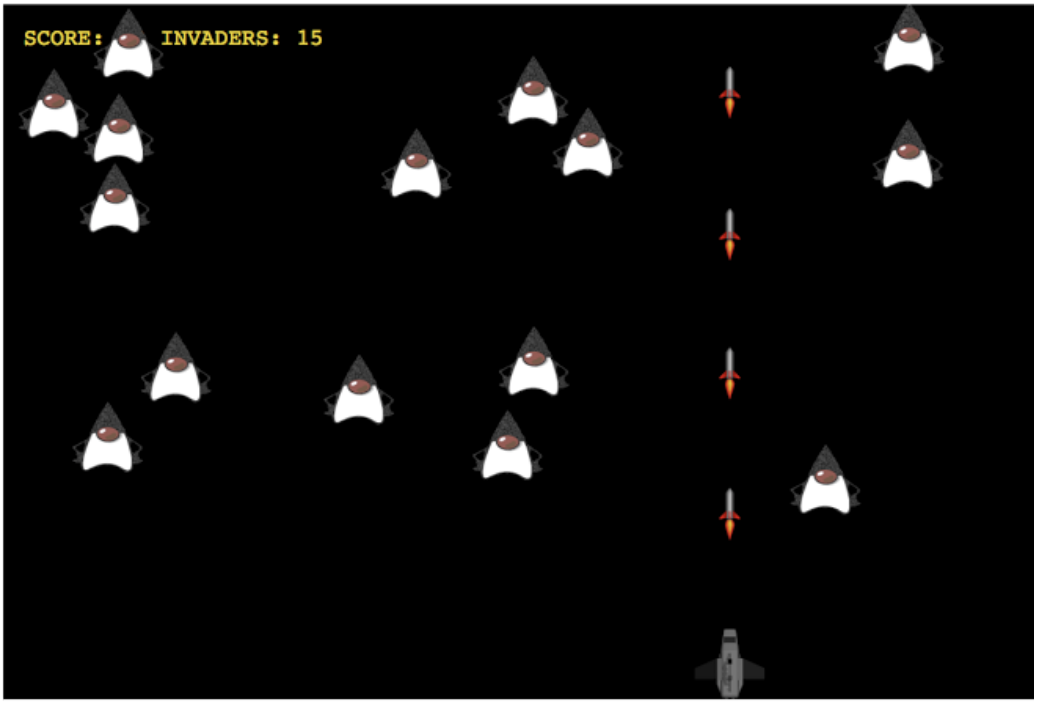

Figura 2. Captura de pantalla. Juego en curso.

Figura 3. Captura de pantalla. Juego perdido.

Figura 4. Captura de pantalla. Inspección de elemento durante el juego en el navegador Safari.