ASL - Lenguaje embebido en Haskell para la creación de Animaciones y Motion Graphics

Juan Bautista Figueredo[0009-0002-6815-7090]

Universidad Nacional de Rosario, Argentina
juan.b.figueredo01@gmail.com

Resumen Este trabajo presenta un lenguaje de dominio específico ASL (*Animation Specification Language*), diseñado para generar animaciones bidimensionales.

ASL es un lenguaje interpretado, imperativo, estáticamente tipado y secuencial. Estas características convierten al lenguaje en una alternativa sencilla, intuitiva y segura, para escribir animaciones de una manera rápida y eficaz.

La expresividad del lenguaje utilizado como lenguaje anfitrión, Haskell, resultó muy conveniente para la implementación del lenguaje. La reutilización de su sintaxis clara, sistema de tipos estático, entorno monádico, librerías y otras partes del lenguaje permitieron desarrollar de manera rápida una implementación robusta.

Los usuarios de ASL podrán construir y operar imágenes y acciones para describir con ellas animaciones completas.

Keywords: Lenguaje De Dominio Especifico, Animaciones, Haskell, Lenguaje Embebido

ASL - Haskell embedded language for Animation and Motion Graphics creation

Abstract. This work presents ASL (*Animation Specification Language*), a domain specific language designed to generate two-dimensional animations.

ASL is an interpreted, imperative, statically typed and sequential language. These characteristics make it a simple, intuitive and safe alternative to write animations quickly and efficiently.

The expressiveness of the host language, Haskell, proved highly convenient for the implementation of the language. The reutilization of its clear sintax, static type system, monadic environment, libraries and other parts of the language allowed the quick development of a robus implementation. ASL enables users to construct and combine images and actions to describe complete animations.

Keywords: Domain specific language, Animations, Haskell, Embedded language

1 Introducción

Desde la integración de los gráficos por computadora en el mundo creativo, los diseñadores de animaciones han optado por utilizar estas nuevas herramientas para potenciar y facilitar la tarea de crear animaciones. De aquí surge el concepto de *Motion Graphics* (Gráficos en Movimiento), que si bien su definición puede ser ambigua, refiere principalmente al uso de estas animaciones generadas por computadora en sectores como el diseño gráfico, el entretenimiento y la educación.

En el mundo de las animaciones y la cinematografía, la unidad mínima de contenido se conoce como *frame* (o *cuadro*). Un *frame* es una imagen. La exposición consecutiva de *frames* genera una ilusión de movimiento, y, si se varía la velocidad con la que se exponen los distintos *frames*, se obtienen distintos resultados en cuanto a la *suavidad* de la animación. Una animación con más *frames por segundo* (*fps*, *frames per second*) resulta más suave que una con una velocidad de *frames* menor. La frecuencia de *frames* (*framerate*) de una animación moderna puede variar entre 30 y 60 *fps*, implicando que para un segundo de animación, deben existir al menos 30 *frames*. En la animación tradicional, el artista debe dibujar cada uno de estos *frames* para poder describir de manera correcta el movimiento que se intenta representar. Usualmente se suele dividir este trabajo en dos etapas: la etapa del *keyframing* (definición de *frames* clave, o *keyframes*) y la etapa del *inbetweening* (definición de *frames* entre *keyframes*) (Harold Whitaker, 1981) (Williams, 2001).

La primer etapa refiere a la definición de ciertos *frames* claves en la animación, como *frames* que definen poses o ubicaciones particulares de los objetos a animar. Una vez definidos los *keyframes*, se procede con la etapa del *inbetweening*, que es el proceso de creación de todos los *frames* intermedios entre los *keyframes*, logrando la ilusión de movimiento.

La animación generada por computadora introdujo un cambio significante al flujo de trabajo de un animador: ahora las computadoras pueden encargarse de realizar la interpolación entre dos *keyframes*, atenuando la carga de trabajo del artista.

Existen numerosos lenguajes de programación de uso general que posibilitan la creación de animaciones. Entre ellos se encuentran C++, Python, Javascript, Haskell, etc. Estos lenguajes cuentan con librerías especificas que permiten que el usuario realice esta tarea, valiéndose de la infraestructura interna de cada lenguaje en particular.

El problema radica en que para poder empezar a crear estas animaciones, uno debe primero conocer el lenguaje subyacente, hecho que ralentiza y obstaculiza la tarea del artista. No sólo ocurre que cada lenguaje maneja de manera distinta la generación gráfica, sino que lenguajes como Haskell, con sintaxis declarativa y paradigma no secuencial, presentan dificultades extras.

ASL (*Animation Specification Language*) fue creado con el foco en el diseñador de animaciones, y con el fin de facilitar el proceso de animación. Al ser un lenguaje de dominio específico, su sintaxis y semántica fueron diseñadas a medida, asemejándose lo más posible al flujo de trabajo tradicional de animación, pero aprovechando las ventajas que los gráficos generados por computadora ofrecen. El lenguaje propone un sistema en donde el usuario realiza una definición de distintos objetos –como imágenes, acciones, colores, escenas y animaciones– partiendo de unidades primitivas y combinándolas con distintos operadores específicos para cada objeto.

Para animar una imagen, se debe especificar la evolución en el movimiento de la misma: las acciones que esta realizará y el orden en que serán realizadas. Esta idea es similar a la generación de *keyframes*, donde sólo se indica el resultado esperado y no se describen los movimientos entre dichos resultados. Este flujo de trabajo, continuo y enfocado en el resultado visual, se asemeja más a la tarea de animar manualmente que a la de programar.

Este último punto refleja la diferencia clave entre ASL y los demás lenguajes utilizados para la creación de animaciones: la cercanía que un programa en ASL tiene con la descripción explícita de una animación. La naturaleza secuencial del lenguaje permite al usuario realizar animaciones complejas, con múltiples objetos independientes en movimiento, en pocas líneas de código.

En este trabajo se presentará parte de la implementación del lenguaje, mostrando las decisiones de diseño, representación de los datos y el uso del mismo.

El código completo en Haskell de este lenguaje de domino específico se encuentra disponible en un repositorio de GitHub[1].

2 Generación de Animaciones

Programas como *Adobe After Effects*, comúnmente utilizados para la creación de *Motion Graphics*, establecen una cierta metodología para esta tarea. Primero, el usuario debe, ya sea importando desde su dispositivo o utilizando herramientas de generación de formas nativas del programa, establecer los objetos que se utilizarán. Una vez creado un objeto, el animador debe describir cómo este se moverá por la escena, determinando, por ejemplo, un movimiento hacia una coordenada específica en la escena de trabajo. Para determinar este movimiento, simplemente se puede "avanzar" en la línea de tiempo y ubicar al objeto en la posición deseada (Chris Meyer, 2010).

No es difícil ver que se han generado dos *keyframes*: Uno en el cuadro donde se introdujo el objeto y otro en donde se especificó su posición final. Al momento de ejecutarse, el programa se encarga de generar todos los *frames* entre los *keyframes* generados. Cabe destacar que en este ejemplo, el usuario no determinó la velocidad a la cual se debe mover el objeto, sino la posición que este debe tener al transcurrir una porción de tiempo –indicada con el avance de la línea de tiempo–. Esto deja en manos del programa determinar la velocidad del movimiento y consecuentemente la cantidad de *frames* a generar.

2.1 Imágenes, Acciones y Animaciones

En el lenguaje implementado, existen tres tipos importantes para la construcción de animaciones: Las **imágenes**, las **acciones** y las **animaciones**.

Las **imágenes** representan los elementos que serán animados: toda animación se realiza sobre una imagen. Las imágenes en ASL se pueden construir de manera personalizada combinando imágenes primitivas, como círculos, rectángulos, triángulos y polígonos regulares.

[1] https://github.com/JuanFigueredo-c/ASL

Las **acciones** se representan en el lenguaje como una transformación en el estado de una imagen. Las acciones se dividen en dos grupos: las de **transformación**, que modifican características como la escala o el ángulo de rotación, y las de **traslación**, que modifican la posición en la escena de un imagen.

Para poder definir acciones, se cuenta con una serie de acciones primitivas, como la acción `move`, que traslada la imagen a una posición dada o la acción `scale` que se utiliza para cambiar la escala de una imagen. Estas acciones primitivas, al definirse, también deben llevar información sobre la **duración** de la transformación en sí. Esto permite al ASL interpolar las velocidades en los cambios de las propiedades de la imagen.

Del mismo modo que las imágenes, las acciones cuentan con operadores específicos para combinarlas y crear nuevos comportamientos. Algunos de estos operadores permiten la ejecución en paralelo de dos acciones o la ejecución en un bucle finito de una acción. Cabe destacar que una acción no está ligada a una imagen en particular, sino que define un movimiento genérico.

El efecto de una acción varía según la animación y el estado en el que se encuentra al ejecutarse la acción. Un ejemplo de esto es el siguiente: suponga que existe una imagen `A` en el punto (0,0) y una imagen `B` en la posición (10, 0), y se define una acción `x` como un desplazamiento a la posición (10,0) en 5 segundos (`Move {10.0, 0.0} 5.0`). Si se aplica la acción `x` a la imagen `A`, la imagen se desplazará al punto (10,0) en 5 segundos. Sin embargo, si se aplica la acción a la imagen `B`, la imagen permanecerá estática durante la duración de la acción. Esto es porque la imagen ya estaba en la posición especificada en la acción, y el estado del mismo no cambia en la duración de la acción.

Una **animación** se define al agregar un elemento de tipo imagen a la escena definida al inicio del programa. Mediante la expresión `place` se pueden especificar las propiedades iniciales que tendrá esta imagen. Estas incluyen la posición en la escena –Relativa al centro geométrico de la imagen–, la escala, y el ángulo de rotación inicial. Esta construcción resulta en el *keyframe* inicial de la animación. Las animaciones se nutren de las acciones que se vinculan mediante el comando `update`. El cambio en las propiedades de la animación que estas implican se puede pensar como una definición de un nuevo *keyframe*. De este modo, a medida que se vinculan acciones a una animación, se definen distintos *keyframes* de la animación final.

2.2 Interpolación de *Inbetweens*

La generación de los *inbetweens* entre cada *keyframe* definido por la adición de acciones a una animación se realiza tomando en cuenta el estado inicial, es decir, los valores de las propiedades de la animación a la hora de aplicar la transformación especificada en la acción, y el estado resultante luego de haber ejecutado la acción. Como cada acción tiene un tiempo de duración asociado, se puede calcular la velocidad a la cual se debe modificar cada característica de la imagen animada para poder cumplir con la especificación final. Por ejemplo, si se define una acción que provoca un cambio de escala de una imagen en un factor `x` durante una cantidad `y` de segundos, la imagen deberá escalarse en un factor de $\frac{x}{y}$ por segundo.

Internamente, toda la animación se representa mediante el uso de un tipo de dato `AnimState` (figura 1) , que encapsula toda la información necesaria para poder graficar

cada cuadro de la animación. Como una animación es una secuencia de acciones, se puede dividir la duración final de la misma en varios segmentos de distinta duración, cada uno representando una acción. De este modo, una acción se puede interpretar como el cambio en las velocidades de cada propiedad de la animación durante un período determinado.

```
  data AnimState = AnimState
{ pic    :: Picture -- Imagen
, rot    :: Float   -- angulo de rotacion
, rotVel :: Float   -- velocidad de rotacion
, pos    :: Point   -- punto actual
, posVel :: Point   -- velocidades en x e y
, scX    :: Float   -- factor de escala en X
, scVelX :: Float   -- velocidad de escala en X
, scY    :: Float   -- factor de escala en Y
, scVelY :: Float   -- velocidad de escala en Y
, angVel :: Float   -- velocidad angular
, orbP   :: Point   -- Punto de giro
} deriving (Show, Eq)
```

Fig. 1. Definición del tipo `AnimState`

Esta lógica se traduce directamente en la interpretación de una acción en ASL. Un **Stepper** es la representación de una acción a nivel de `AnimState`. En pocas palabras, un `Stepper` es una función que cambia algún valor del `AnimState`. Esto permite pensar a una animación como una secuencia de pares (`Duración`, `Stepper`) en donde, a medida se avanza en el tiempo de ejecución, se aplica la función `Stepper` al `AnimState`, cambiando los parámetros del mismo, y esperando que se cumplan otros `Duración` segundos para aplicar el `Stepper` de la acción siguiente.

```
type Stepper = AnimState -> AnimState
```

Fig. 2. Definición del tipo `Stepper`

Para administrar la aplicación de los distintos `Steppers`, una vez que se procesan todas las acciones vinculadas a una animación, se genera una función **Render**. Estas funciones abstraen la secuencia de pares (`Duración`, `Stepper`) y permiten determinar, en cada instante de ejecución de la animación, el estado `AnimState` correcto a graficar. La función `Render` toma el tiempo transcurrido desde el inicio de la animación y un estado actual, y determina cómo evoluciona el mismo. Esto encapsula la lógica de interpolación para cualquier acción compuesta. De esta manera, es fácil ver que, para

una animación, la función `Render` permite determinar las propiedades en cada instante, determinando los *inbetweens*.

```
type Render = Duration -> AnimState -> AnimState
```

Fig. 3. Definición del tipo `Render`

Teniendo todo esto en consideración, una animación queda definida como un par (`AnimState`, `Render`), es decir, el estado inicial de la animación y la función que define cómo evolucionará dicho estado a medida que avanza el tiempo de ejecución.

Con estas definiciones, ASL implementa una estrategia de interpolación continua basada en *keyframes* y velocidades, logrando una animación fluida y determinista a partir de una especificación secuencial y declarativa.

3 Implementación del lenguaje

ASL es un lenguaje embebido en Haskell. Embeber un lenguaje en otro brinda la oportunidad de aprovechar toda la infraestructura del lenguaje anfitrión. El acceso a elementos como el compilador, sistema de tipos, librerías y sintaxis definidos con anterioridad simplifican y aceleran el desarrollo de lenguajes de dominio específico (Gibbons, 2013).

La generación gráfica de las animaciones se logró mediante el uso de la librería de Haskell, *Gloss* (Lippmeier, 2023). Esta librería toma el papel de interfaz entre Haskell y OpenGL, permitiendo la renderización de imágenes y animaciones. Su elección se basó en este último beneficio, que significó una facilitación para el trabajo de implementación del ASL.

El lenguaje implementado también tiene la propiedad de ser interpretado. Esto quiere decir que, al momento de ejecutarse un programa, todas las instrucciones deben evaluarse a medida que se leen. El proceso por el cual pasa cada sentencia al ejecutarse se definió internamente como *pipeline*.

El *pipeline* se divide en tres etapas distinguidas: el parseo, la verificación de tipos y la evaluación de la expresión final. Esta sección explica la implementación del *pipeline principal*.

3.1 Parser

La primer etapa implica el parseo de las sentencias del programa ASL. Este proceso, permite transformar las expresiones de ASL en expresiones del lenguaje anfitrión.

Las sentencias de ASL se dividen en dos tipos, las declaraciones y los comandos. Ambas estructuras sintácticas se definen con los tipos de datos `Decl` y `Comm` respectivamente. Las expresiones (`Exp`) engloban los constructores y operadores de los distintos tipos disponibles en ASL.

```
data Exp = Circle Float Float Fill Exp
         | Rect Float Float Float Fill
             Exp
         | Line Float Float Float Exp
         | Triang Float Float Float
            Exp
         | Polygon Int Float Exp
         -- Expresiones de Imagenes
         | Stack Exp Exp
         | Offset Exp Exp Point
         | Bind Exp Point Exp Point
         | Rot Exp Float
         | RSize Exp Float
         | Paint Exp Exp
         -- Constructores de Acciones
         | Rotate Float Duration
         | Move Point Duration
         | Scale Float Float Duration
         | Static Duration
         | Orbit Point Float Duration
         -- Expresiones de Acciones
         | Seq Exp Exp
         | Par Exp Exp
         | Loop Exp Int
         -- Constructores de
              Animaciones
         | Place Exp Point Float Float
         -- Variables
         | Var Name
         -- Color
         | Color Int Int Int
         deriving (Show, Eq)

data Comm =
        -- Ejecuciones de Acciones
        Update Name Exp
        -- Ejecucion de
             Animaciones
        | Play [Name]

data Decl = Decl Name Type Exp
```

Fig. 4. Definición de los principales constructores del lenguaje, `Exp` (Expresiones), `Decl` (Declaraciones) y `Comm` (Comandos)

Para la implementación del parser utilizó la librería *Happy* de Haskell. Un programa escrito en *Happy* permite recibir la gramática de un lenguaje en forma *BNF* (*Bakus-Naur Form*) y genera un módulo de Haskell con un parser de la gramática.

La razón de la elección de esta librería para este trabajo radica en la facilidad que presenta *Happy* para la generación de un parser. Los demás enfoques, basados en combinadores como la librería *Parsec*, brindan flexibilidad y personalización en la construcción del analizador sintáctico, pero requieren más trabajo para ser implementado. Por otro lado, *Happy* permite definir el parser basándose en la definición casi directa de la sintaxis concreta del lenguaje. Esto facilita la tarea, debido a que la derivación del parser se realiza de manera casi automática.

3.2 Verificación de tipos

La segunda etapa del *pipeline* de procesamiento de una expresión en ASL es la verificación de tipos. Las sentencias de ASL están estáticamente tipadas. Esto quiere decir que el usuario debe especificar de manera explícita, los tipos de las variables que se declararán.

El sistema de tipos es fundamental para la detección de errores semánticos en el momento previo a la evaluación de la sentencia. Esta garantía permite asegurar una correcta utilización del dominio de trabajo (imágenes, acciones y animaciones). Por ejemplo, la verificación de tipos permite evitar la incorrecta asignación de una expresión de acciones a una variable de imágenes, o la vinculación de acciones a objetos que no son animaciones.

El respaldo que brinda la verificación de tipos sobre la corrección en la construcción de los términos permite, una vez pasada la verificación, descartar internamente la necesidad de estos tipos. Esto permitió, en etapas siguientes del *pipeline*, poder tratar a cada

expresión como una expresión semánticamente correcta y por lo tanto facilitar el desarrollo de los evaluadores.

ASL propone cuatro tipos distintos: colores (`color`), imágenes (`image`), acciones (`action`) y animaciones (`anim`). Cada expresión, tanto constructores como operadores, resultan en alguno de estos tipos. De esta manera la verificación se implementa determinando, para una sentencia dada, la valuación del tipo de cada una de las expresiones que la componen. Si estas son aptas para ser operadas, y el tipo de la sentencia coincide, luego se puede asegurar que la instrucción está bien tipada.

```
tcDecl :: MonadASL m => Decl -> m ()
tcDecl (Decl _ t b) = expect t b

tcComm :: MonadASL m => Comm -> m ()
tcComm (Play ans) = mapM_ tcAnims ans
    where tcAnims e = do eTy <- getTy (Var e)
                         if eTy /= AnimT
                         then failASL $ typeErr (Var e) AnimT
                              eTy
                         else return ()
```

Fig. 5. Fragmento del verificador de tipos

3.3 Evaluación de Declaraciones y Comandos

El tercer y último paso del procesamiento de las sentencias implica la evaluación de las expresiones ya verificadas sintácticamente por el parser y semánticamente por el sistema de tipos.

Como se explicó anteriormente, las sentencias se dividen en dos grupos fundamentales: las declaraciones y los comandos. Ambos grupos tienen una semántica completamente distinta y se evalúan de diferente manera.

El primer grupo contiene a la gran mayoría de expresiones que se encontrarán en un programa ASL tipo. Las declaraciones permiten al usuario definir objetos como imágenes, acciones, animaciones, colores y escenas. La idea importante de las declaraciones es que alimentan el entorno del programa, agregando información sobre los valores y los tipos de las variables definidas.
Dentro del grupo de las declaraciones, existe una distinción entre las *declaraciones de escena* y las *declaraciones de variables*.

Una declaración de escena es utilizada para definir las dimensiones y color del fondo donde una animación se llevará a cabo. La particularidad de las declaraciones de escena proviene del hecho que sólo puede existir una única declaración de escena en un mismo programa ASL. Lo que es más, dicha declaración debe ser la primer sentencia del programa. La evaluación de una declaración de escena implica simplemente escribir la

```
Scene Float Float #r,g,b   -- Declaracion de escena
Name : Type = Value        -- Declaracion de variable
```

Fig. 6. Sintaxis de las declaraciones

información respectiva a la escena en el entorno global del programa. Esto se utiliza a la hora de renderizar la escena con *Gloss*.

Las declaraciones de variable se utilizan para definir variables y asignarles un valor. Estas expresiones permiten definir colores, imágenes, acciones y animaciones. Cada variable es única durante la ejecución de un programa. La evaluación de una declaración de variable representa un proceso más complejo que la declaración de escena.

Al momento de procesarse la declaración, la información de la variable, el tipo y la expresión se almacenan en el contexto del programa ejecutado.

La idea detrás de este proceso es traducir, eventualmente, todos los objetos desde su representación interna a una representación equivalente utilizando las herramientas que provee *Gloss*. Para los colores y las imágenes, este proceso puede realizarse de manera casi directa, dado que la interfaz de *Gloss* provee la opción de crear colores y objetos gráficos de tipo `Picture`, como se observa en la figura 7. Para expresiones de acciones, la traducción resulta en la generación de las funciones `Steppers` y para las animaciones, la generación del estado `AnimState` inicial.

```
translateColor :: Exp -> Color
translateColor (AST.Color r g b) = makeColorI r g b 255

translateImg :: Exp -> Picture
translateImg (Circle rad thick fill col) =
    let col' = translateColor col
    in  case fill of
            Full    -> color col' $ circleSolid rad
            Outline -> color col' $ thickCircle rad thick
```

Fig. 7. Fragmento de traducción de expresiones de imágenes y colores a *Gloss*

Para evitar tener que realizar el trabajo de traducción para todas las declaraciones, más allá de si son utilizadas finalmente en una animación que se renderizará o no, se propuso un sistema de evaluación de declaraciones perezosa (*lazy*). Esto quiere decir, que las declaraciones de colores, imágenes, acciones y animaciones sólo se traducirán en el momento en el cual se requiera su representación de *Gloss* para graficarse. De este modo, la evaluación se retrasa, y el procesado en el *pipeline* de una declaración se reduce a encapsular la expresión en un valor artificial (`Value`, figura 8) y almacenarlo en el entorno del programa, vinculándolo con el nombre de la variable declarada.

Cabe destacar que la representación de la evaluación de una expresión de tipo animación, resulta en un valor `An Exp [Exp]`. Los argumentos del constructor `An` rep-

```
data Value = I Exp          -- imagen
           | Ac Exp         -- accion
           | An Exp [Exp]   -- animacion
           | C Exp          -- color
           deriving (Show, Eq)
```

Fig. 8. Definición del tipo `Value`, que encapsula la evaluación de las declaraciones

resentan la expresión utilizada para la definición del primer estado de la animación, determinado con el operador `place`, y una lista de expresiones. Esta última está destinada a almacenar todas las evaluaciones de las expresiones de acciones utilizadas para agregar movimientos a la animación mediante el comando `update`, que se analizará más adelante.

El segundo grupo concentra a todas las sentencias que producen algún cambio sobre los objetos definidos con antelación. Ya sea desde agregar o vincular acciones a animaciones, a ejecutar dichas animaciones.

En ASL actualmente existen 2 comandos, el comando `update` y el comando `play`.

Sintácticamente, el comando `update` se representa como un operador infijo `'<<'`, que opera una animación y una acción. Su uso es el siguiente: `Anim << Action`

Permite vincular una acción a una animación. Esto es, agregar la acción a la secuencia de acciones que la animación ejecutará.
El comando `update` requiere que la animación a operar esté definida previamente mediante una expresión `place`, y por lo tanto, el primer argumento es una variable de tipo `Anim`.

La evaluación del comando `update` requiere la evaluación de la expresión de acción a vincular. Como la animación que recibirá la acción ya fue declarada, existe una variable de tipo `Anim` en el entorno que corresponde con la variable utilizada en el comando. Como se explicó con anterioridad, al evaluarse dicha declaración, se almacena en el entorno un valor `An Exp [Exp]` vinculado al nombre de variable utilizado. Dicho esto, la evaluación entonces se reduce a agregar al entorno, en particular al valor asociado a la variable de animación, el valor resultado de la acción a vincular mediante el comando en sí.

El comando `play` tiene la siguiente sintaxis: `play [Anim1, Anim2, ... , AnimN]`.
La sentencia no solo es obligatoria en cualquier programa ASL, sino que debe ser la última sentencia de un programa. Este comando indica qué animaciones se ejecutarán. El orden en el cual se escriben las animaciones dentro del comando `play` afecta la forma en la que se renderizan las animaciones. Las animaciones se apilarán una sobre otra siguiendo el orden de izquierda a derecha en el que fueron agregadas como argumento al comando `play`. Esto es, las animaciones de la derecha estarán por encima de las animaciones de la izquierda.

La ejecución de este comando resulta en la ejecución y renderización de las animaciones definidas a lo largo del programa, y especificadas como argumento del comando. La evaluación este tipo sentencias se divide en dos etapas.

La primer etapa se trata sobre evaluar todas las animaciones argumento del comando. Este trabajo sigue con la idea explicada en la sección 2. Primero, se traduce, efectivamente, todos los valores resultado de la evaluación de cada animación y sus acciones vinculadas. Este proceso recursivo permite traducir a términos de *Gloss* todos lo componentes necesarios para la generación de la animación. Este paso puede pensarse como la verdadera evaluación de los términos definidos en las declaraciones de variables. Una vez obtenido el estado `AnimState` inicial y la secuencia de pares (`Duracion`,`Stepper`), se procede a calcular y definir la función `Render` asociada a la animación.

La segunda etapa implica la integración definitiva de la interfaz provista por *Gloss* para la generación de gráficas animadas. La función principal que propone *Gloss* para la creación de animaciones es la función `animate`. Esta función recibe tres argumentos, la definición de la pantalla donde se animará, el color de fondo y una función que, dada la cantidad de segundos desde el inicio de la animación, retorne una imagen del tipo de *Gloss*, `Picture`.

Los primeros argumentos, se derivan directamente del resultado de la declaración de escena, y dichos datos están almacenados en el entorno.

El último argumento se construye a partir de la definición de cada función `Render` para cada animación definida. Cada función `Render` toma la cantidad de tiempo desde que comenzó la respectiva animación y retorna el `AnimState` resultante, mientras que la función `producer` (`producer :: AnimState -> Picture`) transforma un estado de una animación en una imagen de *Gloss*. Estas imágenes resultantes se combinaron entre sí para formar una imagen única mediante la función `pictures` de *Gloss*, que cumple este propósito.

El resultado de la función `animate` es la presentación de la animación para el usuario.

3.4 Entorno y Mónadas

El núcleo de la implementación del lenguaje se apoya sobre el uso de mónadas para el correcto manejo de un entorno interno global mutable y efectos secundarios como la propagación de errores en tiempo de ejecución (Hutton, 2016).
La mónada principal, `MASL`, se define como la composición de tres mónadas. Estas son:

La mónada `StateT` permite manejar los cambios que se realicen sobre el entorno, en este caso modelado con el tipo `Env`. En este, se almacenan tanto las variables declaradas a lo largo del programa como los valores finales de los procesamientos de las escenas y las animaciones.

La mónada `ExceptT` brinda la posibilidad de capturar y propagar los posibles errores generados durante la interpretación de las sentencias, como el uso de variables no definidas o el incorrecto uso de tipos. Esta capa es clave a la hora de brindar una garantía con respecto a la consistencia del estado del programa.

Por último, la mónada `IO` habilita la posibilidad de generar los efectos secundarios necesarios para la visualización de las animaciones generadas con *Gloss*.

```
data Env = Env
{ vars     :: [(Name, (Type, Value))]
, cantVars :: Int
, scene    :: Scene
, anims    :: [(AnimState, Render)]
}
```

Fig. 9. Entorno principal `Env`

```
type MASL = StateT Env (ExceptT Error IO)
```

Fig. 10. Definición de mónada `MASL`

4 Conclusión y Trabajos Futuros

Este trabajo tuvo como objetivo desarrollar ASL, un lenguaje de dominio específico embebido en Haskell orientado a la creación de animaciones bidimensionales. A lo largo de su desarrollo, se logró definir una sintaxis clara, un sistema de tipos estático, una semántica operacional segura y, mediante la utilización de la librería *Gloss*, una representación gráfica fluida. El lenguaje fue desarrollado como trabajo práctico final para la materia *Análisis de Lenguajes de Programación*, lo que acotó el alcance del lenguaje a los contenidos de la materia. No obstante, la implementación del lenguaje permitió no solo asentar los conocimientos aprendidos, sino que sentó las bases para su evolución hacia un proyecto de mayor envergadura.

En este sentido, existen diversas líneas de trabajo futuro que pueden mejorar tanto la experiencia del usuario como la calidad en la ejecución de las animaciones. Entre estos trabajos se encuentra la adición de métodos para descargar en formatos como `.mp4` o `.gif` las animaciones especificadas, la implementación de animaciones interactivas con entrada de señales externas que modifiquen su comportamiento en tiempo de ejecución, y la ampliación de la sintaxis y del sistema de renderizado para soportar animaciones en tres dimensiones.

Bibliografía

Chris Meyer TM (2010) Creating Motion Graphics with After Effects, 5th Edition. Routledge

Gibbons J (2013) Functional programming for domain-specific languages. In: ZsÃ³k V, HorvÃ¡th Z, CsatÃ³ L (eds) Central European Functional Programming - Summer School on Domain-Specific Languages, Springer, LNCS, vol 8606, pp 1–28, DOI 10.1007/978-3-319-15940-9_1, URL http://link.springer.com/chapter/10.1007/978-3-319-15940-9_1

Harold Whitaker JH (1981) Timing for Animation. Focal Press, Oxford

Hutton G (2016) Programming in Haskell, 2nd edn. Cambridge University Press, USA

Lippmeier B (2023) gloss: Painless 2d vector graphics, animations and simulations. URL https://hackage.haskell.org/package/gloss

Williams R (2001) The Animator's Survival Kit: A Manual of Methods, Principles, and Formulas for Classical, Computer, Games, Stop Motion, and Internet Animators. Faber and Faber