Interacción y manipulación de estructuras de datos complejas en VizHaskell

Grado en Ingeniería Informática

Facultad de Informática

UNIVERSIDAD
COMPLUTENSE
MADRID

Trabajo de fin de Grado

David Bolaños Calderón

Lidia Marcela Flores Tuesta

Madrid, junio 2016

Director: Manuel Montenegro Montes

Finis coronat opus,

Nunca te rindas

Autorización de difusión y utilización

David Bolaños Calderón

Lidia Marcela Flores Tuesta

Los abajo firmantes, matriculados en el Grado en Ingeniería Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir con fines académicos, no comerciales y mencionando expresamente a sus autores el presente Trabajo de Fin de Grado: “Interacción y manipulación de estructuras de datos complejas en VizHaskell ”, realizado durante el curso académico 2015-2016 bajo la dirección de Manuel Montenegro Montes en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM, a depositarlo en el Archivo Institucional E-Prints Complutense con el objetivo de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Madrid, Junio 2016

David Bolaños

Lidia Flores

Manuel Montenegro

Agradecimientos

En especial, queremos agradecer a nuestro director Manuel Montenegro por su plena confianza depositada en nosotros y su gran disponibilidad a la hora de ayudarnos en cualquier problema que se nos presentaba. Gracias Manuel.

En general, queríamos agradecer a todos aquellos profesores que nos han acompañado a lo largo de todos estos años, que con gran esfuerzo en sus enseñanzas hemos logrado llegar hasta aquí.

Y por supuesto, agradecer a nuestras familias todo su esfuerzo, apoyo y cariño para que salgamos adelante. Sin ellas esto no hubiera sido posible y por ello ahora somos lo que somos.

David Bolaños y Lidia Flores

Dedicatoria

Este proyecto va dedicado principalmente a mis padres, Ana y Ezequiel, y mis hermanos, Manuel y Roberto, quienes me convencieron para estudiar una carrera y creyeron en mí sin dudar nunca, dándome todo el apoyo siempre que lo necesitaba. Podéis estar orgullosos como yo lo estoy de vosotros por quienes sois para mí. Con mucho amor y cariño.

A mis abuelos, Isabel y Manuel, quienes siempre se preocupan por mí y a los que podré contar con orgullo lo que he logrado. Os quiero.

A mis tíos y primos, quienes sé que también quieren lo mejor para mí.

A Lidia, mi mejor amiga y compañera, con la que he pasado prácticamente toda mi vida en la universidad y siempre ha confiado en mí como yo lo hago con ella.

A Samper y Hanjel, mis mejores amigos, quienes siempre están ahí.

A Carlos y Julia, unos amigos increíbles, con quienes también he pasado estos largos años.

Y por último, a otras personas y amigos con los que siempre paso buenos momentos (Andrián, Nikola, Toni, Fulsi...); a las que en algún momento han estado conmigo, como mis antiguos compañeros; y a la UCM, la FDI y a los profesores que me han enseñado lo que sé, logrando que haya llegado hasta donde estoy ahora.

Gracias, esto va por todos vosotros.

David Bolaños

Dedicatoria

Este trabajo se lo quiero dedicar principalmente a mi madre Edith, esa gran persona que siempre ha estado ahí y que sin ella esto jamás lo hubiera conseguido. Gracias Mamá.

Mi hermana Rossangel que me ha apoyado en todo y que ha confiado siempre en mí y que yo no puedo estar más orgullosa de ella.

A mi familia en general, que siempre me da consejos y que nunca me han dejado caer. Sobre todo, para Elsa, Héctor y Yoli que son tan importantes para mí como yo para ellos.

A esa persona que, aunque no esté aquí, siempre ha sido uno de mis pilares más grandes y que desde allí donde esté, está super orgullosa de mí. Va por ti queridísima abuela Marcela.

A Carlos esa persona que jamás podría olvidarme de él, porque decirle lo mucho que lo quiero es poco. Es esa persona que ha estado conmigo en todos estos años y que, sin él, casi seguro que no estaría donde estoy y espero que siempre sea así.

A David, ese gran compañero fiel que hemos ido siempre juntos tanto en las dificultades como en las aventuras, siempre estaré para todo lo que necesites.

Julia, Adrián y esos amigos especiales que son la familia que elegimos, gracias por compartir todos los buenos y malos momentos, me habéis demostrado que sois la mejor familia que pudiera tener.

Y para terminar también quiero dedicárselo a esas personas que me han aceptado en su familia y me han tratado como si siempre hubiera sido miembro de ella. Esa familia que me ha demostrado lo mucho que me quiere y que yo también a ellos, gracias por todos esos consejos y siempre querer lo mejor para mí. Paloma, Antonio S, Antonio SE, Rosa, Ana, Antonio, Mabel y Maria Rosa, muchas gracias por todo vuestro apoyo y amor.

Esto va por todos y cada uno de ellos, porque este trabajo es gracias a vosotros. Os quiero.

Lidia Flores

Índice

Resumen

Nuestro trabajo se ha centrado principalmente en la mejora y extensión de una aplicación web llamada VizHaskell, la cual fue creada en otro proyecto anterior. La finalidad de esta herramienta es ser utilizada en el ámbito educativo para el aprendizaje del lenguaje Haskell y, por ello, incluye diferentes mecanismos para facilitar la programación de este lenguaje, a diferencia de otras que están destinados a un uso más serio o profesional.

Entre sus características están la de incluir un mecanismo de gestión de usuarios en el que cada usuario dispondrá de su propio espacio de trabajo, permitir la creación de proyectos con la que los usuarios pueden tener de una manera organizada los módulos que vayan realizando, incluso proporcionar una visión original a la hora de obtener los resultados de su trabajo y poder entenderlos mejor, mediante el uso de representaciones gráficas como tablas o escenas 3D totalmente manejables y manipulables. Todo esto es posible sin la necesidad de instalar nada más allá de un navegador compatible con HTML5.

Entre las mejoras añadidas están la inclusión de dos nuevas representaciones (tablas asociativas y escenas tridimensionales), la capacidad de anidar complejas estructuras de datos y la capacidad de poder manipular las estructuras de una forma sencilla mediante el uso de conocidos elementos como son los menús contextuales, además de otras mejoras que quizás no son tan llamativas, pero no por ello menos importantes, como la incorporación de funcionalidades relacionadas con los *backreferences*.

Todo esto podemos verlo en detalle en la memoria del proyecto.

Palabras clave

Haskell, VizHaskell, estructura de datos, tipo de datos, representación gráfica, interactividad, aplicación web, programación funcional, escenas 3D, manipulación de estructuras de datos.

Abstract

Our work has mainly focused on the improvement and extension of a web application called VizHaskell, which was created in a previous project. The purpose of this tool is to be used in the educational area for learning Haskell language, which includes different mechanisms to make the programming of this language easier, unlike other tools that are intended for more serious or professional use.

Among its features are an user management tool, in which each user will have his own workspace, and the creation of projects, in which the users can develop their own modules in an organized way. It even provides an original approach to data visualization, in which programmers can get the results of their work and understand them better, by using graphical representations such as fully manageable and modifiable tables or 3D scenes. All this is possible without the need to install anything beyond a browser that supports HTML5.

Added enhancements are the inclusion of two new representations (associative tables and three-dimensional scenes), support for nested complex structures of data, and support for easy data structure manipulation, through the use of known elements such as context menus, as well as other improvements, which are not so conspicuous but no less important, such as the incorporation of features related to *backreferences*.

We shall describe this in detail in the following project report.

Keywords

Haskell, VizHaskell, data structure, data type, graphical representation, interactivity, web application, functional programming, 3D scenes, data structure manipulation.

Capítulo 1

Introducción

En este primer capítulo expondremos las razones que nos motivaron a la hora de realizar este proyecto. También describiremos los objetivos que queríamos alcanzar y la estructura de los capítulos del presente documento.

1.1 Motivación

El desarrollo de Internet fue un hito que revolucionó el mundo de la informática y las telecomunicaciones de una manera espectacular. De hecho, aún lo sigue haciendo hasta el punto en que no hay ámbitos de la vida que no se relacionen de alguna manera con internet. El entretenimiento, la vida social, la noticias, etc. Entre todos estos contextos podemos citar el educativo, que ha adquirido en la última década una notable presencia en las redes, debido a las nuevas herramientas de soporte al aprendizaje (por ejemplo, entornos educativos virtuales como *Moodle*) y a la proliferación de cursos masivos en línea (MOOCs). Esto posibilita el aprendizaje a distancia tan solo mediante el uso de un dispositivo con conexión a Internet y un navegador web.

El sector del aprendizaje en el desarrollo de software ha sido uno de los primeros beneficiados en esta evolución, al ser el software la propia herramienta que ha colaborado a llevar a cabo estos avances. Hace unos años, el aprendizaje de nuevas tecnologías y lenguajes de programación se realizaba con extensos manuales, muy engorrosos y aburridos. Ahora, existen multitud de tutoriales, manuales y cursos a distancia que abarcan una gran variedad de lenguajes de programación (Java, Python, Scala, etc.). Muchas de estas herramientas son interactivas, permitiendo al estudiante visualizar sus progresos.

Estas herramientas *online* también han sido aplicadas al desarrollo de software, donde podemos encontrar auténticos entornos de desarrollo que no tienen nada que envidiar a los típicos *IDE*s de escritorio y en los que podemos desarrollar (y ejecutar) código de gran variedad de lenguajes haciendo uso de la *nube.* Algunos

ejemplos los podemos encontrar en herramientas como *Cloud9*[1] o *Codiad*[2], soportando otros mecanismos como son el control de versiones automático o la posibilidad de desarrollar de forma colaborativa. Otro tipo de herramienta, ya más enfocado al ámbito científico, puede ser *Jupyter Notebook*[3]. Esta proporciona un entorno web interactivo para ejecutar fragmentos de código en *Python*, que además podemos acompañar de gráficos y elementos enriquecidos que ayuden en las explicaciones y temas que se desarrollen.

A pesar de esta diversidad de este tipo de herramientas, sus usos van destinados a la realización de proyectos complejos y profesionales, alejándose de la sencillez necesaria en herramientas para fines educativos. Este fue uno de los motivos importantes a la hora de desarrollar y mejorar *VizHaskell.* Se trata de un entorno de desarrollo *online* centrado fundamentalmente en el lenguaje Haskell, el cual quizás puede ser complicado de aprender, principalmente si uno no ha tenido ocasión de conocer el paradigma de programación funcional en el que se basa. Por otro lado, los intérpretes (como *GHCi*) son muy incómodos de utilizar ya que sus respuestas son generalmente proporcionadas mediante farragosas cadenas en texto plano, muchas veces muy difíciles de interpretar a simple vista, como son el caso de estructuras en árbol. *VizHaskell* surgió con la idea de suplir en lo posible esta necesidad, ofreciendo curiosos e innovadores mecanismos a la hora de mostrar y experimentar con estructuras de datos mediante gráficos, basándose en la idea de "*una imagen vale más que mil palabras"* y facilitando en gran medida la interpretación de las respuestas, algo a tener en cuenta sabiendo que *VizHaskell* es una herramienta de aprendizaje destinada a usuarios poco experimentados.

Conociendo todo esto, vimos interesante la idea de poder trabajar en un proyecto para mejorar VizHaskell, haciéndola de una mejor herramienta de tal forma que otras personas pudieran utilizarla con fines didácticos y adquirir grandes conocimientos sobre lenguaje Haskell y las estructuras de datos.

1.2 Objetivos

Anteriormente, nuestros compañeros Charo Baena y Roberto Aragón realizaron un excelente trabajo en la elaboración de esta aplicación web. Sin embargo, como ocurre con otras herramientas de cualquier otro tipo, siempre hay *algo más* que puede ser añadido para mejorar cualquier aspecto, ya sea visual o funcional. En este sentido, VizHaskell puede dividirse en dos partes, la dedicada a la parte del cliente, de la cual los usuarios hacen uso, y la parte del servidor, en la que se apoya la parte cliente. Es en la primera donde principalmente ha estado

[1] https://c9.io/
[2] http://codiad.com/
[3] http://jupyter.org/

dirigido nuestro trabajo, con objetivos generalmente destinados a la mejora y extensión de la aplicación mediante nuevas funcionalidades de las que podrían sacar partido cualquier usuario que haga uso de esta aplicación.

Uno de los objetivos más importantes que nos planteamos, dando título a nuestro proyecto, fue la interacción con las representaciones gráficas de las estructuras de datos en las que se trabaja en *VizHaskell*, en la que se permitiría la manipulación de esos datos mediante expresiones que pueda configurar el programador de Haskell y hacer uso de ellos mediante elementos interactivos que la gran mayoría conoce, como puede ser menús contextuales o formularios. Todo ello se integra de una manera eficaz pero sencilla teniendo siempre en mente la usabilidad y facilidad de uso de la aplicación. En definitiva, este objetivo va destinado a mejorar la interacción más allá del ámbito visual en el que, por ejemplo, si hacemos uso de los árboles, podemos pinchar sobre los nodos para contraer y expandir sus nodos.

Por otra parte, puesto que anteriormente *VizHaskell* disponía de un solo tipo de representación, vimos que podía ser de gran utilidad la incorporación de más tipos (como pueden ser tablas asociativas, ampliamente utilizadas en la representación de datos en cualquier área), además de relacionar todos esos tipos entre sí mediante el anidamiento de estructuras para hacerlas aún más complejas, aumentando así las posibilidades. Cabe decir que estos objetivos (añadir más tipos por un lado y permitir su anidamiento por otro) son los primeros con los que iniciamos la marcha para posteriormente trabajar con la introducción de interactividad y así ya incorporarlos en todas las representaciones gráficas existentes.

Por último, nos propusimos mejorar la consola añadiendo más funcionalidades que le permitiera acercarlo aún más a una que tenga los mecanismos de los que dispone un intérprete real, como la posibilidad de usar un historial mediante referencias o el funcionamiento de los denominados *backreferences*.

En conclusión, los objetivos principales que planteamos son los siguientes:

1. Añadir dos nuevos tipos de representación: tablas y escenas tridimensionales.
2. Permitir el anidamiento de estructuras de datos en los diferentes tipos que las admitan: de esta forma, se pueden construir complejas estructuras de datos.
3. Añadir mecanismos que permitan interactuar con las representaciones gráficas para manipular los datos de las estructuras, tanto en el tipo de estructura que ya estaba disponible (árboles), como en los nuevos que incorporaremos.
4. Mejorar la consola con mecanismos típicos de intérpretes y nuevas técnicas de programación: para ello se usan los denominados

backreferences, disponiendo de una funcionalidad u otra según el contexto en el que se use dentro de la aplicación.

Como otros objetivos ya más secundarios, y si el tiempo del que disponemos nos lo permite, nos dedicaremos a realizar pequeñas mejoras tanto visuales como funcionales que pudieran surgir durante el desarrollo del proyecto.

1.3 Plan de trabajo

En este apartado describiremos las fases realizadas a lo largo del proyecto, además de mencionar los capítulos y secciones en los que se abordan:

1. **Selección de la metodología de desarrollo y tecnologías.** De cara al desarrollo, debíamos disponer de un método de trabajo además de un plan de tareas a seguir lo más fielmente posible. También debíamos escoger las principales tecnologías que íbamos a utilizar en el desarrollo de los objetivos. Estos aspectos lo veremos comentados en los apartados 3.1 y 3.3 del *Capítulo 3 Metodología y herramientas de desarrollo*.
2. **Instalación y configuración de la herramienta.** Una vez realizado el plan de trabajo, procedimos a seguirlo, comenzando con la instalación de VizHaskell en un entorno virtual y su correcta configuración para establecer las comunicaciones con el entorno físico. Esto lo veremos explicado en el apartado 3.2 del *Capítulo 3 Metodología y herramientas de desarrollo*.
3. **Estudio de la arquitectura existente en la aplicación y familiaridad con la misma.** Puesto que partíamos de un proyecto anterior, debíamos realizar un estudio de los diferentes componentes que de los que consta la aplicación, como puede ser la arquitectura de la que se dispone o los códigos fuentes existentes, además de familiarizarnos con el uso de la aplicación, de cara al futuro desarrollo. Esto lo abordaremos en el Apartado 2.2 del *Capítulo 2 Antecedentes*.
4. **Elaboración de pruebas.** Dado que íbamos a necesitar ejemplos para la realización de pruebas durante la implementación de los nuevos tipos, procedimos con un mecanismo que nos pudiera servir para tal propósito. Esto lo conoceremos como *Mocks* dentro del apartado 4.5 del *Capítulo 4 Contribuciones*.
5. **Incorporación de las tablas asociativas.** Una vez preparado el entorno, comenzamos con la elaboración de las tablas. Los detalles se encuentran en el apartado 4.1.1 dentro del *Capítulo 4 Contribuciones*.
6. **Incorporación de las escenas tridimensional.** Sus aspectos los encontramos en el apartado 4.1.2 dentro del *Capítulo 4 Contribuciones*.
7. **Posibilitar el anidamiento de estructuras.** Ya terminados los nuevos tipos de representaciones, nos dispusimos con el desarrollo de los

mecanismos necesarios para este objetivo. Lo abordaremos en el apartado 4.2 dentro del *Capítulo 4 Contribuciones*.

8. **Incorporación de la interactividad y manipulación de las estructuras.** Los aspectos relacionados con el desarrollo de este objetivo están disponibles en el apartado 4.3 del *Capítulo 4 Contribuciones*.
9. **Desarrollo de los *backreferences*.** Las características de los mismos están disponibles en el apartado 4.4 del *Capítulo 4 Contribuciones*.

Introduction

In this chapter we will discuss the reasons for us to engage in this project. We also describe the goals we wanted to achieve and the structure of the chapters of this document.

1.1 Motivation

Internet development was a milestone that revolutionized the world of computing and telecommunications in a spectacular way. In fact, still does to the point that there are no life aspects that are not related to the Internet in any way. Entertainment, social life, news, etc. Among all these contexts we can cite the educational one, which has acquired in the last decade a significant presence in the networks due to the development of new learning support tools (e.g. virtual learning environments such as Moodle) and the spread of massive online courses (MOOCs). This enables distance learning by only using a device with an Internet connection and a Web browser.

The learning sector in software development has been one of the first beneficiaries in this evolution, as the software tool itself has helped to carry out these advances. A few years ago, learning new technologies and programming languages was done with extensive, cumbersone and boring manuals. Now, there are many tutorials, manuals and distance learning courses covering a wide variety of programming languages (Java, Python, Scala, etc.). Many of these tools are interactive, allowing the student to visualize his progress.

These online tools have also been applied to the development of software, where we can find actual development environments that have nothing to envy from the typical desktop IDEs, and where we can develop (and execute) code of a great variety of languages by using the cloud. There are some examples of these tools, such as *Cloud9* and *Codiad*, supporting other mechanisms, such as automatic version control or collaborative development. There are other types of tools, more devoted to the scientific field, such as Jupyter Notebook. The latter provides an interactive web environment to run code fragments in Python. We can also

accompany graphics and enriched elements that help in the explanations and issues being developed.

Despite the diversity of these tools, their uses are destined to the accomplishment of complex and professional projects, away from the simplicity necessary in tools for educational purposes. This was one of the main reasons to develop and improve VizHaskell. Vizhaskell is an online development environment essentially focused on the Haskell language, which perhaps can be difficult to learn, especially if one has not had the opportunity to learn the functional programming paradigm on which it is based. On the other hand, the interpreters (e.g. GHCi) are rather cumbersome to use because their responses are generally provided by verbose strings in plain text, often very difficult to interpret at a glance, such as those representing tree structures. VizHaskell emerged with the idea of fulfilling this need as much as possible, offering curious and innovative mechanisms when we want to show data structures by means of graphical representations, basing on the idea of a picture is worth a thousand words, and greatly making the interpretation of the answers easier, something to keep in mind given that VizHaskell is a learning tool intended for users with little experience.

Given all this, we were interested in the idea of working on a project to improve VizHaskell, making it a better tool, so that other people use it for educational purposes and in order to acquire great knowledge about the Haskell language and data structures.

1.2 Objectives

Previously, our teammates, Roberto Aragón and Charo Baena, did an excellent job in the development of this web application. However, as with other tools of any kind, there is always more that can be added to improve any aspect, either visual or functional. In this regard, VizHaskell can be divided into two parts, one dedicated to the client, which is what users use, and another dedicated to the server that supports the client. It is the former to which our work has been mainly aimed, with general objectives devoted to improving and expanding the application through new functionality that could benefit any user who use this application.

One of the most important objectives that we consider, and the one that serves as the title of our project, was the interaction with the graphical representations of the data structures that are shown in VizHaskell, The manipulation of these data would be allowed by using expressions given by the Haskell programmer, in addition to the use of known interactive elements, such as contexts menus or forms. All this is integrated in an effective and simple way, always keeping in mind

the usability of the application. Definitively, this objective is intended to improve the interaction beyond the visual elements in which, for example, in the case of trees, we can click on their nodes to contract and expand them.

Later, since VizHaskell provides only one type of representation, we think that it could be useful to incorporate more types (such as associative tables, widely used in the representation of data in any area), as well as to relate all these types together by nesting structures to increase its complexity. It must be said that these objectives (add more types on the one hand and allow their nesting on the other) are the first with which we started the project, and later. We carried out the addition of interactivity in all existing graphic representations.

Lastly, we planned to improve the console by adding more functionality in order to provide it those mechanisms of actual interpreters, such as the possibility of using a history through references or operate with backreferences.

In conclusion, the main objectives are the following:

1. Add two new types of representation: tables and three-dimensional scenes.
2. Allow nesting data structures in different types that support it: in this way, it is possible to build complex data structures.
3. Add mechanisms to interact with graphical representations to manipulate data structures, both in the type of structure that was already available (trees) and the new ones that will be added.
4. Improve console with the typical mechanisms of other interpreters and new programming techniques. In this case backreferences are used, whose functionality depend on the context in which they are used within the application.

If we have time, the secondary objectives that may arise during the project will be devoted to doone some smaller visual and functional improvements.

1.3 Work plan

This section will describe the steps undertaken throughout the project, in addition to mentioning the chapters and sections which are explained:

1. **Selection of development methodology and technologies.** In the development we should have a working method in addition to a plan of tasks to follow as faithfully as possible. We also had to choose the main technologies we would use in the development of objectives. These aspects we will see discussed in sections 3.1 and 3.3 of *Chapter 3 Metodología y herramientas de desarrollo*.

2. **Installation and configuration of the tool.** Once the work plan was sketched, we proceeded to follow it, starting with the installation of VizHaskell in a virtual environment and its correct settings to establish communication with the physical environment. We shall explain this in section 3.2 of Chapter 3 *Metodología y herramientas de desarrollo*.
3. **Study of the existing architecture in the application and become familiar with it.** Since we have undertaken a previously started project, we needed to study he different components comprising the application, such as its architecture and its available source code, in addition to become familiar with how the application is used, in order to address future development. We will discuss this in Section 2.2 of *Chapter 2 Antecedentes*.
4. **Testing**. Since we would need examples of testing during the implementation of the new types, we proceeded with a mechanism that we could serve that purpose, namely a mock, which will be introduced in Section 4.5 of Chapter 4 *Contribuciones*.
5. **Incorporate associative tables.** Once we had prepared the environment, we started with the development of the tables. Details are in section 4.1.1 in Chapter 4 *Contribuciones.*
6. **Incorporate three-dimensional scenes.** *Details are found in section 4.1.2 in Chapter 4 Contribuciones.*
7. **Allow nesting structures.** Having completed new types of representations, we got ready to develop the necessary mechanisms for this purpose. We will address this in section 4.2 in *Chapter 4 Contribuciones.*
8. **Incorporate interactivity and manipulation of structures.** Issues related to the development of this objective are available in paragraph 4.3 of Chapter 4 *Contribuciones*
9. **Backreferences development.** The characteristics of these are available in section 4.4 of Chapter 4 *Contribuciones*.

Capítulo 2

Antecedentes

En el ámbito del desarrollo de software existen varios tipos de paradigmas de programación. Cada uno de ellos determina la metodología utilizada y las técnicas empleadas. Entre ellos podemos encontrar el paradigma imperativo, basado en términos que definen el estado del programa y sentencias que varían dicho estado con el fin de resolver un problema para conseguir una solución, en contraposición al paradigma declarativo, cimentado en la declaración de un conjunto de condiciones que describen el problema y detallan su solución.

Dentro de la programación declarativa nos centraremos en particular en el paradigma funcional, el cual está basado en la evaluación de expresiones puramente matemáticas donde se verifica la propiedad de *transparencia referencial*: el significado de una expresión depende solamente del significado de sus subexpresiones. De esta manera, la evaluación de una expresión no conlleva efectos colaterales y no modifica ningún estado, pues el programa carece de él.

Haskell es un lenguaje de programación funcional puro de propósito general con las características anteriormente mencionadas que presenta una serie de ventajas respecto a otros lenguajes:

- **Tipado estáticamente**. Se asegura en tiempo de compilación que las distintas operaciones se realizan sobre datos del tipo correcto.
- **Conciso**. Proporciona un desarrollo de código más corto y elegante que otros lenguajes, lo cual facilita la comprensión y verificación del mismo.
- **Alto nivel**. Permite abstraernos a un nivel superior de los detalles más complejos, como es el flujo del programa o el manejo de la memoria, el cual es automático, pudiéndose el programador centrar en el algoritmo.
- **Modular**. Haskell ofrece la posibilidad de desarrollar programas a partir de módulos ya existentes.

Por otro lado, existen varios inconvenientes:

- **Difícil depuración.** Al ser un lenguaje puro, no se puede imprimir nada sin declarar el tipo de la función que imprime como IO.
- **Interacción compleja.** Haskell no proporciona una interacción sencilla con el intérprete a usuarios poco experimentados, de manera que se necesitan complicados mecanismos, como es el anidamiento de complejas expresiones para la obtención de un resultado y la comprobación directa y correcta del mismo.
- **Visualización compleja del resultado.** En muchas ocasiones Haskell nos ofrece resultados bastante complejos y difíciles de interpretar, sobre todo para los usuarios noveles.

Debido a estos precedentes surgió la idea de crear una aplicación capaz de solventar en lo posible parte de estos inconvenientes, principalmente los relacionados con la visualización. Se trata de **VizHaskell**.

2.1 VizHaskell

VizHaskell (**Baena y Aragón, 2014**) se trata de una aplicación web cuyo principal objetivo es proporcionar una herramienta enfocada para el entorno educativo, facilitando así el aprendizaje a programadores que se inician en la programación funcional. De manera similar a un IDE, se pueden gestionar proyectos con sus correspondientes módulos, así como ejecutar expresiones que puedan hacer uso de ellos.

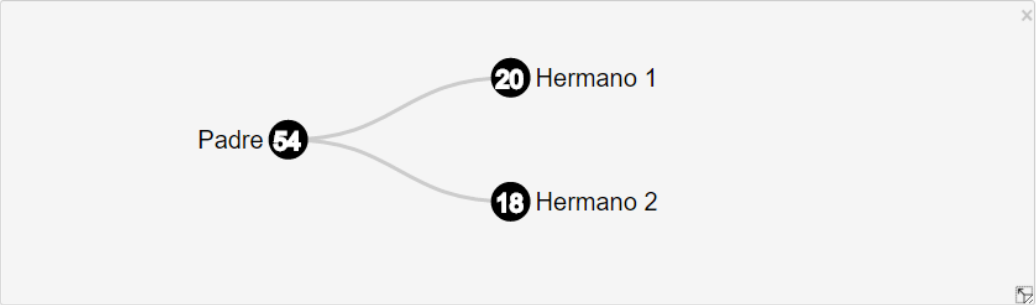
Ya existen herramientas online que simulan intérpretes de Haskell y que son capaces de evaluar expresiones, además de proporcionar pequeños tutoriales para aprender lo esencial, como son *TryClojure*[4] o *TryHaskell*[5] o incluso de disponer de un gestor de proyectos, como el perteneciente a la web de *tutorialspoint*[6], por lo que VizHaskell trata de ir más allá, proporcionando lo que quizás es la parte más destacable de esta herramienta: la visualización de los resultados mediante representaciones gráficas.

Representación gráfica

Ya hemos visto que la representación de datos en Haskell puede llegar a ser muy tediosa, con largas cadenas de expresiones anidadas. VizHaskell proporciona la capacidad de ofrecer los resultados de nuestras expresiones mediante representaciones gráficas interactivas, facilitando en gran medida la labor de poder entender los resultados obtenidos. Esta característica se ofrece

[4] http://www.tryclj.com
[5] https://tryhaskell.org
[6] http://www.tutorialspoint.com/compile_haskell_online.php

mediante una consola, en la que por un lado encontramos la entrada desde la cual los usuarios pueden introducir los comandos y expresiones completas (o los denominados *alias*, previamente configurados dentro de la aplicación y que facilitan el trabajo) y por otro lado encontramos la salida de las expresiones ejecutadas, en la que podemos tener desde los típicos errores, pasando por resultados correctos en texto plano, hasta disponer de auténticas representaciones gráficas, como pueden ser los árboles. Dichas representaciones gráficas tienen un aspecto visual que puede ser definido mediante una hoja de estilos[7], para adecuarse a las necesidades y deseos de los usuarios.



Ejemplo de representación gráfica de un árbol

Usuarios

Por otro lado, cabe mencionar que la aplicación puede ser utilizada por distintos tipos de usuarios, cada uno con unos privilegios u otros dependiendo de su rol. En particular, dispone de los siguientes tres tipos de usuario:

- **Usuarios comunes**. Usuarios a los que va destinada la aplicación. Podrán gestionar sus proyectos y ejecutar los módulos creados o cargados previamente.
- **Administradores de la aplicación**. Disponen de las mismas capacidades que un usuario común, agregando la capacidad de modificar los datos de los demás usuarios.
- **Administrador de servidor**. Dispone de un único usuario el cual se encarga de las altas y bajas de los usuarios en la base de datos.

[7] Lo que comúnmente conocemos como archivos CSS, típica en diseño web.

Página de gestión de usuario para un administrador de la aplicación

2.2 Arquitectura

A continuación, explicaremos la propuesta de arquitectura que utilizaremos en la aplicación. Esta arquitectura viene heredada de la aplicación anterior con la que empezamos a trabajar.
Por este motivo no entraremos en muchos detalles, ya que estos se pueden consultar en profundidad en la memoria del trabajo original.

Partiendo de la idea que lo es esencial de la aplicación es el intérprete de Haskell (GHC), se vio necesario y factible tenerlo instalado únicamente en un mismo sitio, es decir, en un servidor remoto en el cual los equipos usuarios pudieran acceder o comunicarse según lo necesitasen. Con estas premisas, se optó por el uso del paradigma *cliente/servidor.*

2.2.1 Paradigma Cliente/Servidor

El paradigma cliente/servidor es un modelo de aplicación en el cual los denominados clientes demandan la ejecución o realización de una tarea en uno o varios proveedores de recursos o servicios llamados servidores.

Poniendo en práctica estas características, nuestros clientes serían los usuarios que dan uso a la aplicación desde sus equipos interactuando con el servidor. Por ejemplo, cuando el usuario quiere evaluar una expresión ésta se envía al servidor (*petición*), donde se realiza la tarea de evaluación y se responde al cliente con el resultado de la misma (*respuesta*).

2.2.2 Patrón de diseño

Debido a que la parte del cliente requería de la ejecución de tareas con un cierto grado de complejidad pero que no necesitarán del uso del servidor, se decidió por la utilización del patrón Modelo-Vista-Controlador (*Model-View-Controller*) permitiendo además una mantenibilidad del código sencilla.

2.2.3 Propuesta en el Cliente

En este caso el sistema operativo que usen los clientes es independiente del sistema operativo del servidor, pero sí es común a todos los clientes la utilización de navegadores web para poder hacer uso de la aplicación. Es recomendable que dichos navegadores estén actualizados en su última versión para tener una correcta visualización de los elementos gráficos y una buena funcionalidad. También es necesario que el navegador permita la ejecución de lenguaje *Javascript,* pues es el pilar de la aplicación para su funcionamiento, ya que en ella se basan las diferentes librerías y *frameworks* utilizados.

2.2.4 Propuesta en el Servidor

Puesto que se necesitaba de un sistema capaz de realizar procesos complejos de forma simultánea, como almacenar y acceder a datos, ejecutar módulos Haskell mediante GHC y otros tipos de acciones, se debía utilizar un sistema operativo (*SO*) suficientemente potente que permitiera solventar dichas necesidades. Se optó por el uso de Ubuntu, un SO basado en GNU/Linux, libre, completo, estable y suficiente para nuestros propósitos, aunque no se descarta que en un futuro se pueda cambiar por otro con más posibilidades si así es requerido.

Se requería la instalación de *Apache WebServer* en su versión 2.4.6 con los módulos mod_rewrite y mod_proxy activados. Apache se encarga de proporcionar las páginas solicitadas así como de la redirección de puertos mediante *proxys* para la conexión con la base de datos.

El cometido de la aplicación es permitir la creación y ejecución de módulos y comandos de Haskell, por lo que es necesario del uso de un intérprete para tal cometido. Puesto que es inviable que cada cliente tenga que disponer de dicho intérprete, con todo lo que eso conlleva (instalaciones, configuraciones...) la solución era disponer del recurso de manera *online*. Así, los clientes no tienen más que disponer de los medios necesarios para acceder al recurso. Dadas estas circunstancias, el compilador e intérprete escogidos son GHC y GHCi, unos de los más utilizados en la programación funcional con *Haskell*.

Dada la necesidad de disponer de un registro de usuarios del sistema, así como el almacenamiento de sus proyectos con sus correspondientes módulos, se utiliza una base de datos no relacional, en este caso *CouchDB.* Se decidió utilizar este sistema debido a sus características como el uso del protocolo REST, utilizado por la propia aplicación para la solicitud de recursos, y la eliminación de la necesidad de elaborar componentes que permitan operaciones CRUD, pues estas ya están soportadas por su API.

2.2.5 Protocolos

En este apartado se procederá a especificar los diferentes protocolos que se han utilizado en ambas capas.

- **Comunicaciones.** Se ha utilizado el protocolo *HTTP* ya sea para la petición de la página web enviada por el cliente al servidor, así como la solicitud de recursos que se requieran y, por otro lado, el protocolo SMTP (*Simple Mail Transfer Protocol*), protocolo de red utilizado para el envío de correos con el restablecimiento de contraseñas previamente solicitados por los usuarios.
- **Servicios.** Se decidió contar con REST, destinado a la petición del restablecimiento de las contraseñas, así como la solicitud asíncrona de la ejecución de las expresiones y devolución de los resultados de las mismas, utilizándose para ellos el formato JSON (JavaScript Object Notation) dada su sencillez y su afinidad con el lenguaje Javascript.

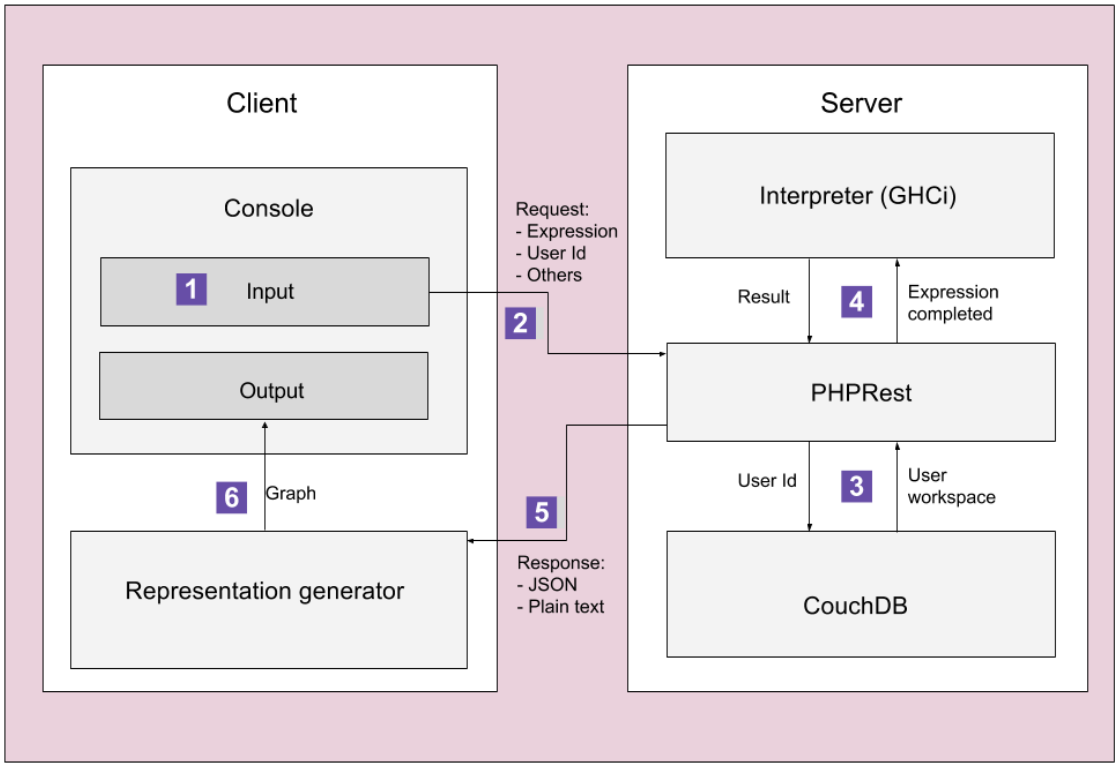
2.2.6 Descripción general de la arquitectura

Para el correcto funcionamiento de la aplicación se ha de disponer de un contrato JSON, el cual es un texto en formato JSON que define la estructura de los datos que se va a representar gráficamente. Este contrato JSON es, por un lado, el modelo que deben seguir los módulos Haskell cuando crean las respuestas mediante el intérprete, y por otro, la estructura usada por las distintas librerías gráficas disponibles en *VizHaskell* para su construcción gráfica, por lo que es lógico que en su estructura se cumplan siempre unos mínimos requisitos para un correcto funcionamiento.

Ya descrito el contrato, el funcionamiento general de la aplicación es el siguiente:

1. El usuario escribe una expresión en la entrada de la consola.
2. Mediante los servicios de la aplicación, se establece una comunicación con el servidor, enviando la expresión a evaluar por el intérprete junto con otros datos necesarios como puede ser el identificador del usuario.

3. Mediante PHP, el servidor obtiene la ruta de los módulos del usuario haciendo uso de la base de datos.
4. Se construye la expresión completa en base a los módulos Haskell correspondientes y se procede con su evaluación obteniéndose el resultado (o un error).
5. Mediante la comunicación previamente establecida, el servidor envía la respuesta con el resultado (ya sea en formato *JSON* o un texto plano) de vuelta al cliente.
6. Ya en el cliente, se realiza el procesamiento oportuno en base a la información de la estructura *JSON* y dibuja en pantalla su resultado como representación gráfica.



Capítulo 3

Metodología

En este apartado procederemos a explicar tanto la metodología aplicada al desarrollo como las herramientas utilizadas para llevar a cabo la realización del proyecto.

3.1 Modelo de desarrollo.

Para la realización del proyecto comenzamos con la descripción de una serie de fases, cada una de ellas con un propósito específico.

1. **Planteamiento de objetivos y tareas.** En esta fase nos dedicamos a marcar los principales objetivos que queremos englobar. Para ello, los integrantes entablamos una reunión para plantear dichos objetivos mediante una lluvia de ideas. También nos encargamos de elaborar un calendario para la realización de las diferentes tareas de las que se compone el proyecto. Esto se hizo mediante un diagrama de *Gantt* para la organización del mismo. *Véase Anexo I.*
2. **Despliegue y aprendizaje de la aplicación.** Una vez conocidos los objetivos, para poder llevarlos a cabo tuvimos que comenzar con el despliegue e instalación de la aplicación y de las herramientas que debíamos necesitar para su desarrollo para luego continuar con el aprendizaje y estudio de la aplicación.
   Para la instalación de la herramienta previamente tuvimos que disponer del entorno de simulación, esto es, la utilización de una máquina virtual en la que instalar el servidor. Por supuesto, no se podía hacer nada sin antes conocer la aplicación, por lo que realizamos una serie de tareas entre las que se encuentran:

   a. Lectura de la memoria del proyecto anterior.
   b. Uso y prueba de la aplicación web tanto desde el punto de vista de un usuario como de un administrador.
   c. Estudio superficial del código desarrollado.

Cabe decir que esta fase también la dedicamos a la instalación de las herramientas necesarias para el desarrollo, como la instalación de un editor de texto o las herramientas para realización de copias de seguridad. En los siguientes apartados entraremos en más detalles.

3. **Desarrollo.** Una vez realizados los pasos previos, ya pudimos comenzar con el trabajo. Puesto que partíamos de un producto final, nuestro trabajo se iba a dedicar principalmente a la mejora y actualización del mismo, añadiendo nuevas funcionalidades para permitir mayores posibilidades de uso. Partiendo del tipo de trabajo que íbamos a desarrollar, decidimos utilizar el modelo evolutivo-incremental para esta fase, la cual constaría de cuatro tareas que se realizaron en orden y de manera repetida hasta la consecución de los objetivos:

    a. **Planteamiento y análisis**. Esta tarea estaba dedicada a analizar la aplicación buscando una nueva funcionalidad que podría ser útil y de interés para los usuarios.
    b. **Búsqueda de información.** Pasamos a buscar la información pertinente para desarrollar la funcionalidad como, por ejemplo, librerías u otras herramientas existentes, que pudieran ayudarnos con lo planteado.
    c. **Desarrollo.** Ya planteado el requisito y elegido el tipo de tecnología a utilizar, se procedió con el desarrollo del mismo.
    d. **Prueba.** En esta última fase nos dedicamos a buscar y corregir posibles errores que pudieran surgir.

4. **Pruebas y documentación.** Todo proyecto necesita una documentación en la que se debe reflejar todos los pasos que se han llevado a cabo para poder realizarlo, y así facilitar un trabajo futuro del mismo. En consecuencia, llevamos a cabo la elaboración del presente documento, en la que dicha tarea constaba de dos fases:

    a. **Borrador de la memoria.** Se procedió con la elaboración de un borrador de la memoria, la cual debería estar en su fase final de redacción. Esta fue entregada a nuestro profesor y director del proyecto para su corrección.
    b. **Memoria definitiva y reunión del material.** Dedicada a corregir los errores encontrados, así como la elaboración y reunión del material para la entrega final.

    De forma paralela a estas dos fases, se llevaron a cabo unas pruebas generales.

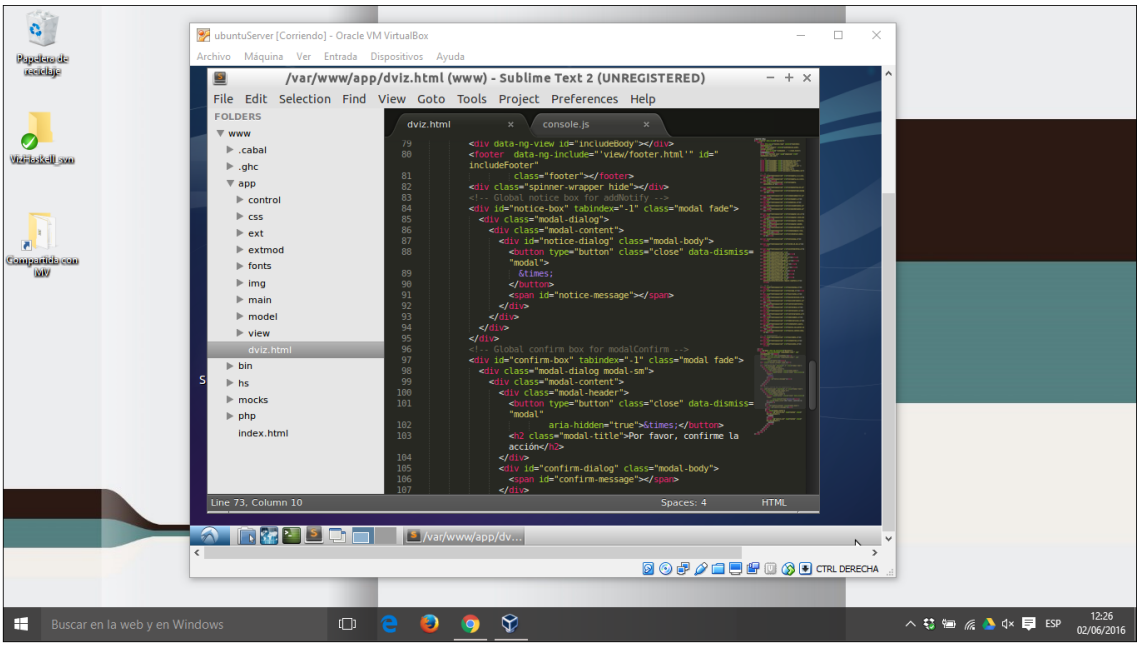
3.2 Entorno de desarrollo

Como se mencionó anteriormente, se utiliza Ubuntu como sistema operativo en la parte del servidor de la aplicación y, debido a la necesidad de un entorno de desarrollo para la realización de pruebas, se optó por simular este sistema operativo mediante máquinas virtuales.

3.2.1 Simulación del entorno: MV con VirtualBox

Dentro de la gran variedad de programas dirigidos a este fin, se decidió optar por la utilización de VirtualBox[8], un software de virtualización con licencia dual (GPL y de pago) que se encuentra en constante desarrollo, es muy maduro y multiplataforma, y se utiliza en la virtualización de sistemas operativos, denominados *invitados* (en nuestro caso, Ubuntu), dentro de otro sistema operativo, llamado *anfitrión* (en nuestro caso, Windows 8 y 10), además de permitir la instalación de los denominados *Guest additions*[9], paquetes software que, tras su instalación en el SO *invitado,* permiten una mayor productividad y una mejor comunicación con el equipo anfitrión mediante la incorporación de ciertos mecanismos como son la integración del ratón o la creación de carpetas compartidas entre ambos sistemas, además de otras ventajas. Por otro lado, VirtualBox también se caracteriza por la capacidad de realizar copias de seguridad de las máquinas virtuales y recuperarlas en caso necesario.

Decidimos utilizar Virtualbox, al ser la herramienta usada anteriormente en el proyecto previo y debido a la familiaridad que tenemos con ella.



[8] https://www.virtualbox.org/
[9] https://www.virtualbox.org/manual/ch04.html

3.2.1.1 Sistema operativo de la aplicación

Una vez instalado VirtualBox para la emulación del servidor, añadimos a Ubuntu unas ciertas características para que facilitaran la realización del trabajo, entre ellas se encuentran:

- **Instalación de un escritorio.** Se instaló el escritorio *LXDE*[10], puesto que es ligero y suficiente para mostrarnos un área de trabajo más amigable e intuitiva, sobre todo para la navegación rápida entre directorios, además de permitirnos instalar herramientas extra.
- **Instalación de un editor de texto.** Se optó por el uso de Sublime Text[11] para modificaciones puntuales y la depuración del código.
- **Guest additions.** Esta instalación era necesaria, ya que nos permite el uso de carpetas compartidas entre la máquina anfitrión y la máquina huésped facilitándonos el traspaso de archivos, como los códigos fuente desarrollados externamente.

3.2.1.2 Sistema operativo para realizar el desarrollo

El ambiente principal de desarrollo fue en el sistema operativo Windows 8 y 10, debido a la experiencia con su manejo. En él se instalaron diversas herramientas para facilitar el trabajo, de las cuales cabe destacar las siguientes:

- **Servidor Apache.** Un servidor web HTTP utilizado en este caso para la comunicación con la aplicación alojada en la máquina virtual. Se vio necesaria la modificación del archivo de configuración de Apache en Windows (*httpd.conf*) para poder crear un *proxy* que permitiera conectarnos con la máquina virtual alojada en VirtualBox.
- **Sublime Text.** Al igual que en el sistema operativo anfitrión, en la máquina de desarrollo se ha instalado este editor de texto que nos permite la instalación de *plugins* para una mayor productividad en la elaboración de código. Ha sido la herramienta principal para este fin.
- **Repositorios.** Herramienta de almacenamiento de archivos, gestión de versiones y realización de copias de seguridad. Para ello hicimos uso de dos ellos, *Apache Subversion* (*SVN)* en el que obtuvimos el código fuente de la aplicación y *Dropbox* para uso propio de archivos e intercambios de documentos.
- **Navegador Web.** Utilización de los navegadores *Mozilla Firefox* y, principalmente, *Google Chrome*. Requerido por razones obvias para el uso y aprendizaje de la aplicación, el testeo y depuración del código,

[10] http://www.lxde.org/
[11] https://www.sublimetext.com/

búsqueda de documentación y ejemplos y realización de la memoria mediante Google Docs, pues nos permite colaborar de manera conjunta y compartir el trabajo elaborado.

- **VirtualBox.** Como ya se explicó anteriormente, usado para la virtualización de Ubuntu. Se tuvo que establecer la redirección de puertos dentro de las opciones de la máquina virtual para una correcta conexión con el servidor, en concreto, de los puertos 9080 y 9022 de la máquina anfitrión a los puertos 80 y 22 de la máquina invitada, respectivamente.

3.3 Tecnologías y lenguajes utilizados

En este apartado procederemos a enumerar y explicar los distintos lenguajes, frameworks y tecnologías que se utilizaron a largo del desarrollo de la aplicación.

- **HTML5**[12]. (*HyperText Markup Language*) Lenguaje utilizado para la elaboración y estructuración páginas web. Utilizado para la construcción de los componentes estáticos de la interfaz de la aplicación, así como su uso en conjunto con Javascript para la creación de los elementos dinámicos.
- **CSS**[13]**.** (*Cascading Style Sheets*) Lenguaje encargado de dar un formato visual a un documento estructurado, en este caso al documento HTML. Se necesitó para adaptar al gusto del desarrollador de la aplicación los efectos visuales que traía *Bootstrap* por defecto, además de usarse también por parte de los usuarios para especificar la presentación de las representaciones gráficas mostradas en VizHaskell.
- **Bootstrap**[14]**.** Framework para diseño de páginas y aplicaciones web. Ofrece un marco de diseño para los elementos *HTML* apoyándose, si es necesario, en *jQuery* para creación de efectos visuales y la adición de nuevas funcionalidades. Se basa en el sistema de rejillas, ofreciéndose así la capacidad de crear páginas web adaptables a distintos tamaños de pantallas, algo realmente útil para *VizHaskell* dada su intención de poder ser usado desde cualquier tipo de dispositivo.
- **AngularJS**[15]**.** Framework MVC en Javascript para su uso en Front-End, desarrollado y mantenido por Google. Facilita la creación y mantenimiento de aplicaciones al poder separar la vista, los controladores y los modelos, además de permitir añadir nuevos módulos que pudieran necesitarse, por lo que fue la elección escogida para el desarrollo de *VizHaskell*. Entre los

[12] https://www.w3.org/TR/html5/
[13] https://www.w3.org/Style/CSS/
[14] http://getbootstrap.com/
[15] https://www.angularjs.org/

módulos podemos mencionar `File-Uploader` para la gestión de subida de archivos o `ngResource`, para la interacción con los servicios REST.

- **GHC/GHCi**[16]. (*Glasgow Haskell Compiler*) Compilador e intérprete, respectivamente, para Haskell. Uno de los más activos en el mundo, en constante desarrollo y de código libre.
- **JavaScript**. Lenguaje de programación interpretado, usado principalmente en navegadores web. Es quizás la parte esencial de la aplicación en el lado del cliente, puesto que es utilizado por los demás componentes.
- **jQuery**[17]. Biblioteca de Javascript, empleada para simplificar la interacción y manipulación de los elementos HTML, creación de efectos y animaciones, manejo de eventos, etc.. Opciones muy útiles para la aplicación dada la dinamicidad de esta. Por otro lado, es una dependencia a su vez de otros frameworks, como *Bootstrap*.
- **JSON**[18]. Formato de texto ligero para el intercambio de datos. Basado en Javascript, es más simple que XML, pero a su vez suficiente en el tratamiento de las respuestas enviadas por el servidor. Por otro lado es utilizado para el formato de texto plano de las representaciones gráficas previo a su tratamiento.
- **PHP**[19]. (*PHP Hypertext Preprocessor)* Lenguaje de programación de uso general para el lado del servidor. En este caso, no es utilizado para la creación de contenido HTML, pero sí para la elaboración de una API REST para el procesamiento de las peticiones AJAX que demandan los clientes, así como su uso para el envío de correo y acceso a la base de datos.
- **D3**[20]**.** (*Data-Driven Documents*) Librería en Javascript que, haciendo uso de tecnologías como SVG, HTML5 y CSS, permite crear infogramas y gráficos dinámicos e interactivos. Aunque por ahora solo es utilizada para las representaciones gráficas de árboles en la aplicación, puede ser utilizada para otros tipos de representación.
- **THREE**[21]**.** Librería en Javascript que, apoyándose en Canvas, SVG, WebGL, o incluso CSS, permite la elaboración de gráficos y escenas en 3D, además de permitir mecanismos con los que movernos e interactuar en dichas escenas.
- **JSCSSP**[22]**.** Librería en JavaScript, robusta y extensible, que permite desde validar texto CSS hasta filtrar sus reglas, así como proporcionar un objeto fácilmente accesible con Javascript.

[16] https://www.haskell.org/ghc/
[17] https://jquery.com/
[18] http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf
[19] http://php.net/
[20] https://d3js.org/
[21] http://threejs.org/
[22] http://www.glazman.org/JSCSSP/index.html

Capítulo 4

Contribuciones

En este capítulo procederemos a explicar detalladamente las nuevas incorporaciones y mejoras en la herramienta VizHaskell. Para ello, hemos realizado una serie de tareas tanto en la parte del servidor como en la del cliente, centrándonos principalmente en esta última,
Intentando cumplir todos los objetivos propuestos inicialmente. Las mejoras van desde la adición de nuevos tipos de representación hasta la interactividad con los mismos, entre otros cambios.

4.1 Tipos de representaciones

Teniendo en cuenta que disponíamos de un solo tipo de representación gráfica (los árboles) nos propusimos la idea de poder añadir nuevos tipos de representaciones, ya que esto sería interesante desde el punto de vista de los usuarios. Estos nuevos tipos enriquecerán la aplicación, ofreciendo más posibilidades a los usuarios para poder entender los resultados de su trabajo y facilitar en gran medida su aprendizaje y/o productividad. Dicho esto, procederemos con la descripción de los nuevos tipos.

4.1.1 Tablas asociativas

Las tablas son herramientas para mostrar información de forma organizada, muy útiles y una de las más utilizadas en todos los ámbitos, sobre todo en el mundo de la informática estando presente de una manera muy importante. Un ejemplo bastante claro son las bases de datos relacionales donde las tablas son la esencia de las mismas.

Elegimos esta representación puesto que una tabla es fácil de interpretar a la hora relacionar diferentes datos de manera pertinente y puede aportar una visión concisa en la percepción de los mismos para los usuarios. En concreto se decidió

trabajar con tablas asociativas, que son aquellas que se caracterizan por disponer de una estructura en la que unos valores están asociados a una *clave o llave* y son útiles para almacenar grandes cantidades de información.

4.1.1.1 Contrato JSON

El contrato JSON es la estructura de datos utilizada para definir las representaciones gráficas en VizHaskell. Para el caso de las tablas asociativas, la estructura general será la siguiente:

```
{
   "repType":"table",
   "label":"",
   "columns": [
       {
           "name": "Columna1"
       },
       {
           "name": "Columna2"
       }
   ],
   "rows": [
       {
           "key":{
               "content”:{
                   "repType":"",
                   "text":""
               }
              "className":""
           },
           "values": [
               {
                   "content”:{
                       "repType":"",
                       "text":""
                   },
                   "className":""
               }
           ]
        }
   ]
}
```

Donde:

`repType:` Campo obligatorio que indica el tipo de representación, siendo en este caso ***`table`***.
`label:` Campo opcional para indicar el título de la tabla.
`columns:` Array opcional en la que se incluyen las columnas de la tabla. A su vez, cada columna puede tener los siguientes campos, ambos opcionales, ya que si no se incluye en el JSON, se pondrán nombres por defecto.

- **`name:`** Indica el título de la columna.

`rows:` Array opcional en la que se incluyen las filas de la tabla. A su vez, cada fila puede tener los siguientes campos:

- **`key:`** Campo obligatorio el cual es la clave de la fila. A su vez, tiene los dos siguientes campos:
    - **`contents`**: campo obligatorio donde se debe incluir de forma obligatoria cualquier estructura de los tipos de representación conocidos en VizHaskell (***`text`***, ***`tree`***, ***`table`*** o ***`scene`***) con sus correspondientes campos.
    - **`className:`** Campo opcional en el que se indica el nombre de la clase con la que se asociará el aspecto visual de la hoja de estilos.

- **`values:`** Campo obligatorio que, mediante un array, representa los valores asociados a la clave. Internamente tiene los mismos campos y funciona de igual manera que el campo **`key`**.

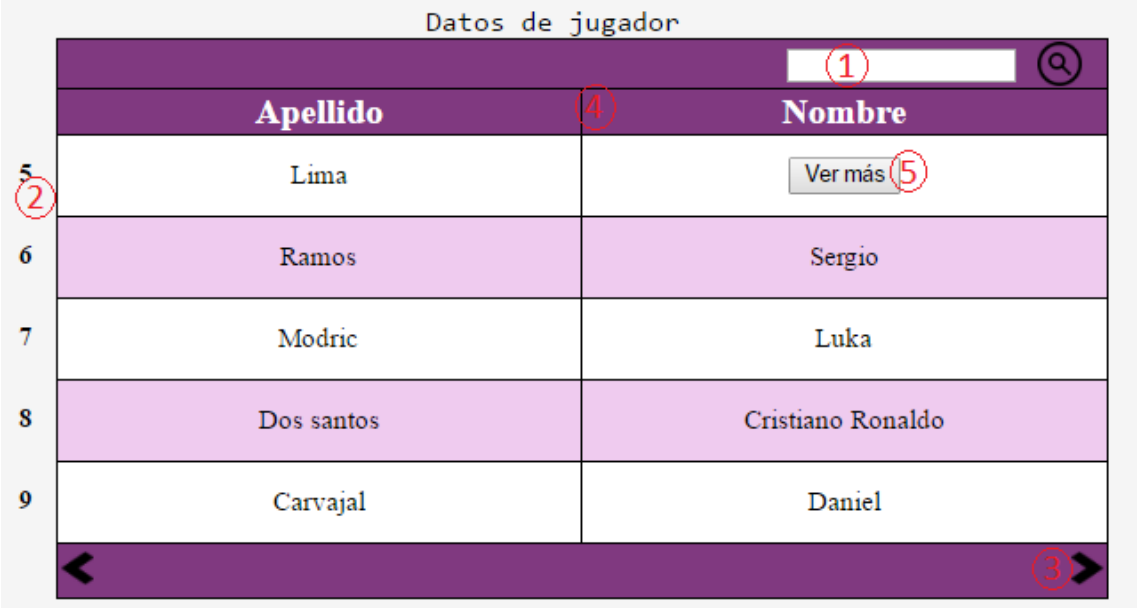
Véase anexo II en el que se muestra un ejemplo sencillo de una representación gráfica de una tabla y su correspondiente JSON.

4.1.1.2 Estructura

Para llevar a cabo la elaboración de la tabla y sus elementos hemos hecho uso íntegramente de la librería *jQuery*. Entre esos elementos podemos encontrar los siguientes:

1. **Un cuadro de búsqueda**. Podemos utilizar el cuadro de búsqueda para filtrar resultados de la tabla.
2. **Indicadores de fila**. Sirve para indicar el número de fila que estamos observando. Esto es útil para que sepamos en qué punto de la tabla nos encontramos en todo momento, ya que de otro modo, por ejemplo, si realizamos una búsqueda, no sabríamos a qué número de fila pertenecen esos resultados encontrados.
3. **Botones de navegación**. Dado que las tablas pueden llegar a tener un tamaño bastante considerable, estos botones nos permiten navegar por la tabla, visualizando poco a poco de una manera cómoda los resultados de la misma.
4. **Cuerpo de la tabla.** Parte de la tabla donde se representa la información de la misma.
5. **Botón *Ver más*.** Este botón que podemos encontrar en las celdas presenta dos funcionalidades donde, una de ellas nos permite visualizar textos largos que de otra forma romperían la estética de la tabla, y la otra muestra los tipos de representación que se le haya dado al valor de la

celda mediante anidamiento. Ambas visualizaciones se mostrarían mediante el uso de ventanas modales. *Véase apartado 4.2 Anidamiento.*



Para obtener un mejor entendimiento del funcionamiento de las tablas, el código está disponible en el script *table.js.*

4.1.1.3 Búsqueda

Ya hemos mencionado que las tablas pueden ser bastantes grandes, por lo que se decidió añadir un mecanismo de búsqueda a la misma para facilitar su visualización. El algoritmo utilizado para poder realizar una búsqueda se basa en la ocultación de las filas en las cuales no haya al menos una celda que contenga en su valor los caracteres escritos en la caja de búsqueda. Este algoritmo se ejecuta en tiempo real a medida que modificamos el cuadro de búsqueda y su complejidad es del orden *O(n)* en el mejor de los casos y *O(nm)*(n = nº de filas y m = nºcolumnas) en el peor.

El nombre de la función que contiene el algoritmo es ***`function doSearch()`*** el cual podemos encontrarlo dentro del script ***`app/main/table.js`***.
Por defecto el cuadro de búsqueda está oculto, por lo que si se desea utilizarlo basta con pulsar sobre la imagen de la lupa mostrándose con un efecto visual. Por el contrario, si se encuentra visible y pulsamos nuevamente, este se ocultará.

4.1.1.4 Estilo

La tabla por defecto se muestra con las tonalidades violetas particulares de VizHaskell, pero si se desea cambiar el aspecto, no tenemos más que indicarlo en la hoja estilo, y siempre que previamente, se haya especificado mediante un nombre de clase (*className*) en el contrato JSON de la tabla. Estas modificaciones sólo son posibles en el estilo de las celdas y las propiedades que admiten en la hoja de estilo son las existentes en el estándar CSS, como sería el caso del color del fondo, la fuente de la letra, los bordes, etc..

4.1.2 Escenas 3D

Actualmente, la evolución tanto en las tecnologías usadas por los navegadores web como el aumento en la capacidad de cálculo y cómputo de los computadores actuales permite sin gran coste la generación de gráficos 3D, por lo que debido a este avance no parece una locura pensar que puede ser interesante la idea de poder mostrar información con una serie de gráficos o estructuras más complejas de las que habitualmente se suelen usar. Esta idea la hemos plasmado usando para ello escenas tridimensionales, de las que hablaremos a continuación.

4.1.2.1 Contrato JSON

Para las escenas 3D la estructura general será la siguiente:

```
{
   "repType":"scene",
   "className":"",
   "lights": [
      {
          "id":"",
          "type":"",
          "className":"",
          "position":{
              "x":"", "y":"", "z":""
          },
          "lookAt":{
             "x":"", "y":"", "z":""
          }
      }
   ],
   "shapes": [
      {
          "id":"",
          "type":"",
          "className":"",
          "measures":{...},
          "position":{
             "x":"", "y":"", "z":""
          },
          "rotation":{
             "x":"", "y":"", "z":""
          }
      }
   ]
}
```

Donde:

`repType:` Campo obligatorio que indica el tipo de representación, siendo en este caso ***`scene`***.
`className:` Campo opcional para indicar el nombre de la clase de la escena que irá en el archivo de la hoja de estilo (*Styles.css*).
`lights:` Campo opcional en el que, mediante un array, se incluyen las luces de la escena. A su vez, contiene las siguientes opciones (todas opcionales salvo el **`id,`** el cual es obligatorio y único dentro de la escena):

- **`id:`** identificador de la luz dentro de la escena, de tipo *String* (un nombre, un número...)
- **`type:`** indica el tipo de luz entre los que se encuentran ***`directional`***, ***`pointLight`***, ***`spotLight`*** o ***`ambientLight`*** (por defecto). En la sección 4.1.2.2.4 se describirá con más detalle la diferencia entre los distintos tipos de luz.
- **`className:`** nombre de la clase de luz que irá en el archivo *Styles.css*.
- **`position:`** objeto que indica la posición de la luz a partir de las coordenadas { x, y, z }. Estos son *doubles* que pueden ser negativos, positivos o cero.

- **`lookAt:`** objeto que indica la posición donde "apuntará" la luz, a partir de { x, y, z } con el mismo formato que para la posición. Esta propiedad sólo está disponible para algunos tipos de luz.

`shapes:` Campo opcional en el que, mediante un array, se incluyen las figuras de la escena. A su vez, los objetos que pueden ser opcionales, contienen los siguientes campos:

- **`id:`** Único y obligatorio. Identificador de la figura dentro de la escena, de tipo *String* (un nombre, un número...)
- **`className:`** Campo opcional para indicar el nombre de la clase de la figura que irá en el archivo *Styles.css*. En los apartados siguientes se explicarán con más detalle.
- **`position:`** Objeto opcional que indica la posición de la figura a partir de las coordenadas { x, y, z }. Estos son *doubles* que pueden ser negativos, positivos o cero.
- **`rotation:`** Objeto opcional que indica la rotación en grados respecto a los ejes de coordenadas { x, y, z }. Estos son *doubles* que pueden ser negativos, positivos o cero.
- **`type:`** Campo obligatorio para indicar el tipo de la figura (esfera, prisma, etc..)
- **`measures:`** Objeto obligatorio que, dependiendo del tipo de la figura, contendrá los parámetros necesarios (y obligatorios) para el mismo. En los apartados siguientes se explicarán con más detalle.

Véase anexo III en el que se muestra un ejemplo sencillo de una representación gráfica de una escena y su correspondiente JSON.

4.1.2.2 Estructura

En este apartado nos dedicaremos a la descripción de las escenas tridimensionales y la explicación de todos los elementos que la componen, aunque si queremos tener una visión más profunda y mejor comprensión de las escenas en *VizHaskell,* se puede acceder al estudio del código en el script ***`app/main/scene.js`***.

Para la creación de las escenas hemos hecho uso de la librería *Three.js* (**Cabello, 2010**), en comparación con otras librerías como son *BabylonJS*[23] o *SceneJS*[24]. Nos decidimos por esta librería dada su popularidad, su constante crecimiento y su implementación basada en módulos, permitiendo excluir aquellas funcionalidades que no fueran imprescindibles y que añaden una

[23] http://www.babylonjs.com/
[24] http://scenejs.org/

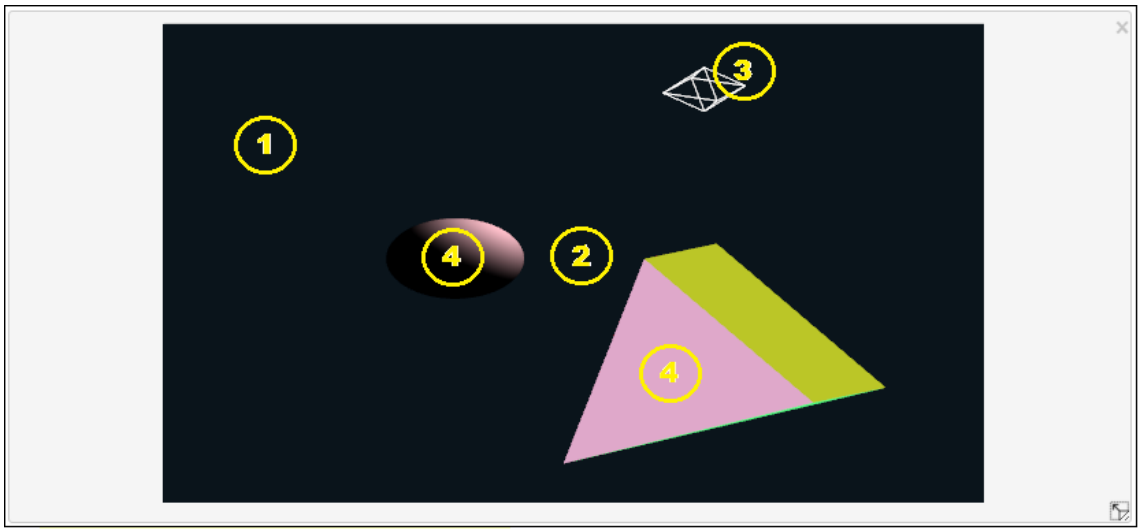
complejidad innecesaria, pero que siempre se pudiera añadir en un futuro si fuera necesario. Por otra parte, dispone de mecanismos para poder crear escenas a partir de estructuras *JSON* predefinidas, pero muy enrevesadas para nuestros propósitos, razón por la cual nos decidimos por la creación de nuestro propio contrato *JSON*, más acorde con la sencillez y coherencia de los demás tipos de representación.

Three.js se desarrolló en el 2010 y en un principio fue escrita en *ActionScript* para finalmente ser traducida a *Javascript*. Inicialmente se utilizaban renderizadores basados en *Canvas* o *SVG*, pero gracias a los avances que se hicieron en los navegadores web, como puede ser la inserción de *WebGL*, *Three.js* pudo aprovecharse de todas las ventajas que esta tecnología aporta y así mejorar su visualización.

Una de las ventajas que aporta *WebGL* es el rendimiento, porque puede hacer un uso de la aceleración por hardware debido a que esta tecnología está basada en *OpenGL*[25] y utilizada por el navegador Web, mejorando y realizando el renderizado de los gráficos en 3D dentro del mismo.

Los elementos de los que se dispone en una escena son:

1. **La propia escena.** En ella se añaden los demás elementos.
2. **Una cámara.** Necesaria para poder visualizar la escena.
3. **Luces.** Necesarias para poder ver los objetos de las escenas.
4. **Figuras.** Son los elementos que “adornan” y dan sentido a la escena.



[25] Especificación que define una API multiplataforma y multilenguaje, utilizada para la creación de gráficos 2D y 3D a la hora de escribir aplicaciones. https://www.opengl.org/.

Para obtener un mejor entendimiento tanto de la estructura como del funcionamiento de las escenas, podemos ver el código fuente en el script *scene.js*

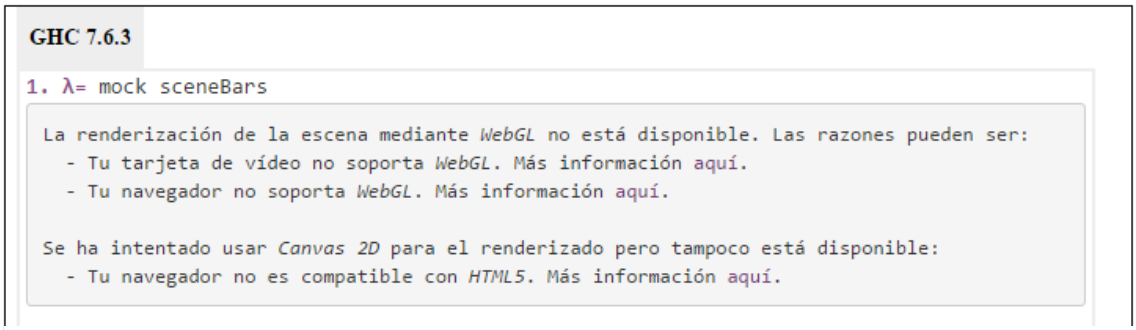
4.1.2.2.1 Renderizado

Las escenas en *THREE* necesitan que sean renderizadas para su correcta visualización. Para ello, se ha de contar con un renderizador que se encargue de esa tarea, es decir, generar las imágenes y animaciones 3D a partir de unos modelos, como pueden ser las figuras y/o las luces. El renderizador se configura en el momento de crear la escena, y para que esta pueda ser visible correctamente, se debe sustentar sobre algún elemento del documento, que en este caso, es sobre el elemento *canvas* de *HTML5.*

Entre los renderizadores que dispone *THREE* encontramos:

- **CanvasRenderer.** Basado en el uso de la *API* de *Canvas 2D*.
- **WebGLRenderer.** Basado en el uso de la *API* de *WebGL*.
- **CSS3DRenderer.** Basado en los efectos 3D mediante *CSS*. No disponible actualmente en la aplicación.
- **SVGRenderer.** Basado en el uso de SVG. No disponible actualmente en la aplicación.

Por defecto, *VizHaskell* utiliza *WebGLRenderer*, por lo que se debe disponer de un navegador, además de una tarjeta de vídeo, compatibles que soporte y tenga activado *WebGL*. Por el contrario, si WebGL no está disponible, como alternativa se utiliza *CanvasRenderer* en su lugar, más lento y pesado que *WebGL* pero que nos puede servir de igual manera para los intereses de los usuarios. Si por alguna razón el navegador es incapaz de utilizar uno de estos renderizadores, las escenas no podrán generarse, mostrándose en su lugar un mensaje de error si intentamos usar este tipo de representación. Estas comprobaciones es lo primero que se realiza al inicio de la función **`addScene`** para así poder detener el procesado de las escenas y evitar la ejecución de instrucciones innecesarias. Las comprobaciones se realizan apoyándose en el script *Detector.js*, desarrollado también por el creador de *THREE* para lidiar con este tipo de situaciones.



Ejemplo de error

4.1.2.2.2 Escena

Una escena se puede definir como la representación visual de una situación, producto de un renderizado. Es decir, una escena funciona a modo de contenedor, donde podemos añadir y extraer, por ejemplo, figuras o formas, cada una con sus propiedades y peculiaridades, además de tener ciertos comportamientos ante determinados eventos.

Una escena puede ser de varios tipos: 2D o 3D, según sus dimensiones, siendo esta última con la que hemos trabajado.

El siguiente fragmento de código es el encargado de crear la escena y su renderizado:

```
1. scene = new THREE.Scene();
2. renderer = new THREE.WebGLRenderer( { antialias: false } );
3. …
4. baseDiv.append( renderer.domElement );
5. renderer.render( scene, camera );
```

Donde:

1. Creación de la escena.
2. Configuración del renderizador a utilizar, en este caso, WebGL. Tambíen es posible utilizar Canvas (`new THREE.CanvasRenderer`) y SVG (`new THREE.SVGRenderer`).
3. Otras instrucciones complementarias, como por ejemplo, establecer el tamaño de la escena.
4. Adición del elemento *canvas* en el que se plasma la escena sobre un *div* perteneciente al DOM.

5. Especificación de la escena y la cámara con la que trabajará el renderizador.

4.1.2.2.3 Cámara

Para poder visualizar la escena, necesitamos algún mecanismo para indicar el punto de vista de la misma. En este caso, *Three.js* pone a disposición el uso de una cámara. Dicha cámara puede ser de varios tipos, según su perspectiva, entre las que se dispone de:

- **PerspectiveCamera.** Cámara cuya proyección se basa en perspectiva cónica.
- **OrthographicCamera.** Cámara cuya proyección se basa en perspectiva isométrica.
- **CubeCamera.** Cámara formada a su vez por seis *PerspectiveCamera.*

Por otro lado, hemos incluido un mecanismo para poder mover la cámara, es decir, movernos a través de la escena permitiéndonos ver con detalle los distintos elementos que lo componen. Para ello, hemos hecho uso del módulo *THREE*, *OrbitControl.js*[26], que modifica la visualización de la escena en base a los eventos del ratón o gestos, si se trata de una pantalla táctil. La siguiente tabla detalla las acciones y sus respuestas sobre la escena:

Acción	Respuesta
Mover el ratón mientras se mantiene pulsado el botón izquierdo.	Variar la posición de la cámara en la escena sin cambiar la posición del punto que enfoca.
Mantener y mover un dedo en una pantalla táctil.	
Mover el ratón mientras se mantiene pulsado el botón central.	Aumentar o disminuir el zoom en la escena respecto al punto de enfoque.
Usar la rueda del ratón.	
Pellizcar o estirar usando dos dedos en una pantalla táctil.	
Mover el ratón mientras se mantiene pulsado el botón derecho.	Variar la posición de la cámara en la escena cambiando también la posición del punto que enfoca.
Pulsar las flechas ← , → , ↑ y ↓ del teclado. (Actualmente no disponible)	
Mantener y mover usando tres dedos a la vez sobre una pantalla táctil.	

[26] https://gist.github.com/mrflix/8351020

El fragmento de código encargado de configurar la cámara y los controles es el siguiente:

```
1. camera = new THREE.PerspectiveCamera( 45 , window.innerWidth / window.innerHeight , 1 , 10000);
2. camera.position.x = 250;
3. camera.position.y = 250;
4. camera.position.z = 250;
5. camera.lookAt( new THREE.Vector3( 0, 0, 0 ) );
6. controls = new THREE.OrbitControls( camera, document, renderer.domElement );
7. controls.addEventListener( 'change', render );
```

Donde:

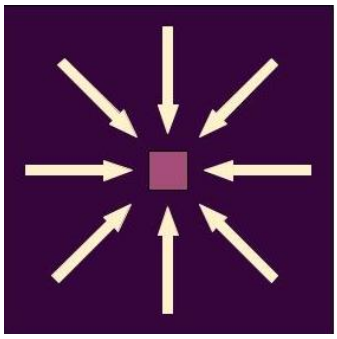
1. Se crea la cámara a partir de su campo de visión, la relación de aspecto y distancia mínima y máxima de visión. En el caso de *VizHaskell*, por ahora, solo es posible la utilización de ***PerspectiveCamera***.
2. Posición de la cámara en la coordenada X de la escena.
3. Posición de la cámara en la coordenada Y de la escena.
4. Posición de la cámara en la coordenada Z de la escena.
5. Posición en la escena donde “mirará” la cámara (punto donde enfoca).
6. Creamos y configuramos el control indicándole la cámara y el elemento HTML del cual detectar los eventos anteriormente descritos.
7. Cuando se genere un evento, volver a renderizar la escena para actualizarla.

4.1.2.2.4 Luces

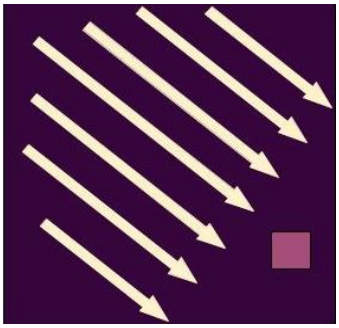
La luz en una escena forma una parte importante de la misma, debido a que sin la existencia de ésta no es posible visualizar nada de su interior. Por ello las escenas de VizHaskell ofrecen la posibilidad de introducir una o varias luces. Si el usuario no incluye ninguna se introduce por defecto una luz ambiental para poder ver correctamente los elementos de la escena. Esto evita la confusión del usuario, puesto que puede pensar que tiene un error en su trabajo realizado, cuando realmente sí que pueden estar presentes las figuras creadas.

Mediante el contrato JSON se puede definir tanto la posición como la dirección de las luces en la escena, las cuales, si no vienen dadas por el usuario, por defecto estarán dirigidas al origen de coordenadas y se colocarán sobre este con una altura de 100. También se define el tipo de luz, entre las que encontramos:

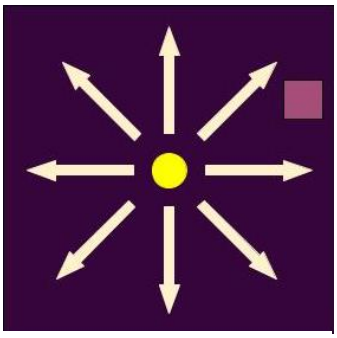
- **AmbientLight.** Este tipo de luz se aplica a todos los objetos de la escena de manera uniforme independientemente de la posición de los mismos. En similitud con la realidad podemos pensar en la luz del sol en un día nublado. Es el tipo escogido por defecto si el usuario no introduce ninguna luz.
- **DirectionalLight.** Luz sin un punto de origen fijo que se dirige de forma paralela a todos los puntos de la escena. De forma análoga a la realidad podemos verlo como los rayos que recibimos del sol que, al ser un punto de luz muy grande, los percibimos como si estos fueran paralelos.
- **PointLight.** Punto de luz que alumbra en todas direcciones. Para poder comprenderlo mejor podemos pensar en el funcionamiento de una bombilla.
- **SpotLight.** Luz que alumbra con un ángulo concreto. Análogamente es comparable con una linterna.



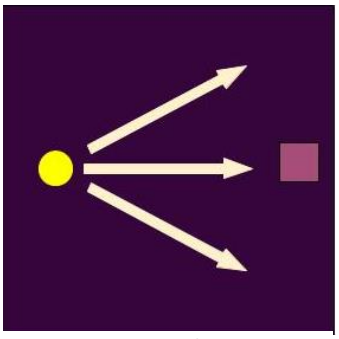
AmbientLight



DirectionalLight



PointLight



SpotLight

El siguiente fragmento de código es el encargado de crear la luz:

```
1. var light = createSpotLight( lightColor, intensity, distance );
2. positionateObject( light, jsonLight, defaultPos );
3. lookAt( light, jsonLight );
4. …
5. scene.add(light);
```

Donde:

1. Creamos una luz del tipo deseado con sus parámetros necesarios. En este ejemplo se crea una luz de tipo *SpotLight* con el color de la luz, la intensidad y la distancia.
2. Se posiciona la luz creada anteriormente. Si la posición no está indicada en el contrato se coloca en una por defecto.
3. Dirigimos la luz hacia el punto indicado en el contrato.
4. Otras instrucciones complementarias.
5. Se añade a la escena la luz creada.

Cabe decir que, por defecto, las luces no se ven en la escena, por lo que se añadido un elemento especial, denominado **`helper`**, para indicar al usuario la posición de las luces que haya insertado.

4.1.2.2.5 Figuras

Las figuras son elementos que sirven para enriquecer y decorar la escena, aunque en el caso de *VizHaskell* también son necesarias para poder obtener estructuras complejas que puedan representar la información deseada, como por ejemplo, un gráfico de barras.

De momento en la aplicación solo están permitidas determinadas figuras básicas pero que a su vez se pueden organizar de tal manera que compongan una estructura que sea capaz de alcanzar el propósito requerido. Al igual que las luces, la posición y la rotación pueden venir definidas en el contrato y en caso que no sea así, se colocará la figura sin rotación y en el origen de coordenadas.

Las posibles figuras son:

- **Paralelogramo.** Cuadrilátero donde los lados opuestos son paralelos dos a dos, aunque solo está disponible para cuadrados y rectángulos. Las medidas requeridas del contrato son:
    - **`sideLength1.`** Longitud del lado 1 y su paralelo siendo un número decimal positivo.
    - **`sideLength2.`** Longitud del lado 2 y su paralelo siendo un número decimal positivo.

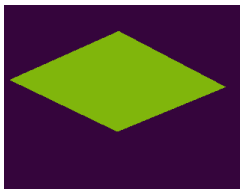
```
"type": "parallelogram",
"measures": {
   "sideLength1": 5,
   "sideLength2": 20
}
```

- **Polígono regular.** Polígono donde sus lados y ángulos son iguales. Las medidas requeridas del contrato son:
    - **`numSides.`** Número de lados de los que se compone el polígono siendo un número natural mayor o igual que tres.
    - **`sideLength.`** Longitud de cada uno de sus lados siendo un número decimal positivo.

```
"type": "regPolygon",
"measures":
{
   "numSides": 5,
    "sideLength":
 200
```

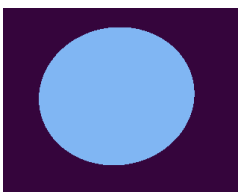
- **Rombo.** Paralelogramo cuyas diagonales son perpendiculares entre sí. Las medidas requeridas del contrato son:
    - **`diag1.`** Longitud de la diagonal uno siendo un número decimal positivo.
    - **`diag2.`** Longitud de la diagonal dos siendo un número decimal positivo.

```
"type": "rhombus",
"measures":
{
   "diag1": 50,
   "diag2": 100
}
```



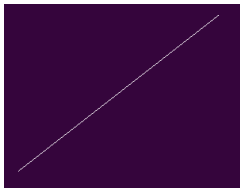
- **Círculo.** Curva cerrada cuyos puntos equidistan del centro. La medida requerida del contrato es:
    - **`radius.`** Radio del círculo siendo un número decimal positivo.

```
"type": "circle",
"measures":
{

   "radius": 100
}
```



- **Línea.** Sucesión continua de puntos que es tratada como un segmento. La medida requerida del contrato es:
    - **`length.`** Longitud de la línea siendo un número decimal positivo.

```
"type": "line",
"measures":
{

   "length": 100
}
```

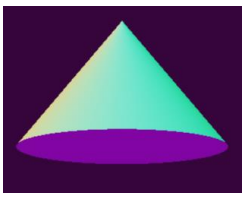


- **Esfera.** Cuerpo geométrico limitado por una superficie curva donde todos los puntos equidistan del centro. La medida requerida del contrato es:
    - **`radius.`** Radio de la esfera siendo un número decimal positivo.

```
"type": "sphere",
"measures":
{

   "radius": 100
}
```

- **Cono.** Cuerpo geométrico formado por una superficie lateral curva y cerrada que termina en un vértice y tiene una base circular. Las medidas requeridas del contrato son:
    - **`radius.`** Radio de la base circular siendo un número decimal positivo.
    - **`height.`** Altura del cono siendo un número decimal positivo.

```
"type": "cone",
"measures":
{
   "radius": 50,
   "height": 100
}
```



- **Cilindro.** Cuerpo geométrico formado por una superficie lateral curva y cerrada y dos bases circulares. Las medidas requeridas del contrato son:
    - **`radius.`** Radio de las bases siendo un número decimal positivo.
    - **`height.`** Altura del cilindro siendo un número decimal positivo.

```
"type": "cylinder",
"measures":
{
   "radius": 50,
   "height": 100
}
```

- **Texto en 3D.** Composición de signos y caracteres en 3D. La medida requerida del contrato es:
    - **`text`.** El texto a mostrar.

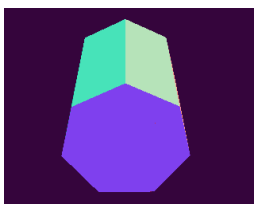
```
"type": "text3D",
"measures":
{
   "text": "Texto de ejemplo"
}
```

- **Pirámide.** Cuerpo geométrico cuya base es un polígono cualquiera y cuyos lados son triángulos con un vértice común. Las medidas requeridas del contrato son:
    - **`base`.** Base de la pirámide formada por una de las figuras planas disponibles y las características de los mismos (**`rhombus`, `parallelogram`, `regPolygon`** y **`circle`**).
    - **`height`.** Altura de la pirámide siendo un número decimal positivo.

```
"type": "pyramid",
"measures":
{
   "base": {
       …
   },
   "height": 100
}
```

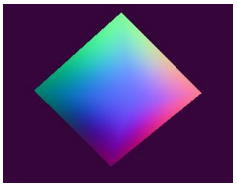
- **Prisma.** Cuerpo geométrico cuyas bases son polígonos y cuyos lados son paralelogramos. Las medidas requeridas del contrato son:
    - **`base`.** Bases del prisma formado por cada una de las figuras planas y las características de los mismos (**`rhombus`, `parallelogram`, `regPolygon`** y **`circle`**).
    - **`height`.** Altura del siendo un número decimal positivo.

```
"type": "prism",
"measures":
{
   "base": {
       …
   },
   "height": 100
}
```



- **Plumbob.** Cuerpo geométrico formado por dos pirámides en espejo con base común. Las medidas requeridas del contrato son:
    - **`sideLength`.** Longitud de cada uno de los lados que forman las bases de dichas pirámides. La altura es calculada automáticamente.

```
"type": "plumbob",
"measures":
{
   "sideLength":50
}
```



El siguiente fragmento de código es el encargado de crear la figura:

```
1. var shape = new THREE.Mesh( geometry, material );
2. positionateObject( light, jsonShape, defaultPos );
3. rotateObject( shape , jsonShape );
4. …
5. scene.add(shape);
```

Donde:

1. Creamos una figura a partir de su geometría[27] y su textura.
2. Se posiciona la figura creada anteriormente. Si no está indicado en el contrato la posición se coloca en por defecto en el origen de coordenadas.
3. Se rota la figura a partir de los ángulos[28] respecto de los ejes de coordenadas indicados en el contrato.
4. Otras instrucciones complementarias.
5. Se añade a la escena la figura creada.

4.1.2.3 Estilo

Una escena se caracteriza por su aspecto visual, de manera que *VizHaskell* ofrece los medios necesarios para poder configurarlo según se requiera. Para ello, al igual que para los demás tipos de representación, se hace uso de una hoja de estilos donde se introducen las clases de los elementos que componen

[27] La geometría son los puntos y aristas necesarios para formar una figura.
[28] La medida de los ángulos utilizada por *THREE* son los radianes, por lo que internamente lo convertimos en grados, requerido en el contrato *JSON*.

las escenas y que están relacionadas con los campos **`className`** introducidos en el contrato. Algunas de estas propiedades no son estándar de la especificación *CSS3*, lo que da lugar a la utilización de unas propiedades propias para *VizHaskell*. Por otro lado, las escenas se sustentan en un componente *canvas,* donde no existen elementos HTML en los que acceder de manera trivial. Por estos dos motivos se hizo necesario usar un intérprete de definiciones *CSS* para poder obtener las propiedades necesarias y así poder utilizarlas en los distintos elementos de los que se compone la escena.

4.1.2.3.1 Propiedades CSS

Las diferentes propiedades CSS que podemos incluir en el archivo *Style.css* son las siguientes:

- **`color.`** Indica el color de las luces, de la textura de las figuras o del fondo de la escena mediante un valor en hexadecimal o usando los *alias* de colores por defecto en CSS. Si no incluimos esta propiedad, o el formato es incorrecto, el valor por defecto para las luces y figuras será **`#FFFFFF`** y para el fondo de la escena, **`#35053C.`**
- **`intensity.`** Intensidad de la emisión de luz, siendo un número decimal mayor o igual que cero. Si no incluimos esta propiedad o el formato es incorrecto, el valor por defecto será **0**.
- **`distance.`** Distancia durante la cual la intensidad de la luz se irá atenuándose linealmente desde su valor máximo hasta **0**, siendo un número decimal mayor o igual que cero. Si no incluimos esta propiedad o el formato es incorrecto, el valor por defecto será **0**[29].
- **`material.`** Tipo de material utilizado para aplicar una textura a las figuras. Los valores admitidos, de tipo *String,* son los siguientes:
    - **`lambert.`** Material a utilizar si se requiere que la textura tenga poco brillo ante una luz.
    - **`normal.`** Material con diversos colores que van cambiando según el ángulo de visión respecto a los puntos de la superficie. Es independiente de la propiedad **`color`** y de la existencia de luces. Es el valor por defecto si no incluimos ningún material o el valor no cumple el formato especificado.
    - **`basic.`** Material con un color constante por toda la superficie de la figura, independientemente de la existencia y propiedades de las luces.

[29] De esta forma, la atenuación de la luz dependerá únicamente del valor de la intensidad.

    - **`phong.`** Material a utilizar si se requiere que la textura con cierto brillo ante una luz, es decir, que genere una *imagen especular*[30].

- **`width.`** Propiedad exclusiva de las *líneas*, sirve para indicar el grosor de las mismas siendo un número decimal mayor que **0**. Si no incluimos esta propiedad o el formato es incorrecto, el valor por defecto será **3**. Esta propiedad no es aplicable si el renderizador usado es *WebGL* y el sistema operativo del usuario es *Windows*, ya que por ciertas limitaciones de los mismos no está disponible, por lo que en ese caso, el valor por defecto será **1**.
- **`font-size.`** Exclusiva de los *textos 3D*, sirve para indicar el tamaño de fuente del texto, siendo un este un número entero mayor o igual que **1**. Si no incluimos esta propiedad o el formato es incorrecto, el valor por defecto será **10**.

4.1.2.3.2 Parseador del CSS

Como ya mencionamos anteriormente, para poder dar estilo a las escenas, es necesario de la existencia de un parseador de texto *CSS*. Para ello, hacemos uso de la librería *JSCSSP* (**Glazman, 2010**) **l**a cual, a partir de un texto *CSS*, nos devuelve un objeto que se puede recorrer para obtener las reglas y sus propiedades con sus correspondientes valores. Sobre este objeto se realiza un filtrado eliminando las propiedades que no sean pertinentes en el ámbito de las escenas. Esto significa que nos quedamos solo con las reglas que contengan alguna de las propiedades mencionadas en el apartado anterior.

En siguiente fragmento de código se encarga de parsear y filtrar el CSS:

```
1. var parser = new CSSParser();
2. var sheet = parser.parse( css, false, true );
3. var rules = sheet.cssRules;
4. var allSceneProps = [ "color", "intensity", … , "font-size" ];
5. rules = filterRules( rules, allSceneProps  );
```

Donde:

1. Se inicializa el intérprete.
2. Se interpreta el texto CSS.
3. Se obtiene el objeto con las reglas del parseado obtenido.

[30] Es una imagen generada como producto de la reflexión de la luz en una superficie especular, que es aquella en la que los rayos incidentes son reflejados con un ángulo igual a la incidencia.

4. Se crea un array con las propiedades disponibles en la escena y sus elementos.
5. Se filtran las reglas, eliminado aquellas no incluidas en el array anterior.

A medida que se va procesando la estructura *JSON*, cuando encontramos el campo **`classname,`** se procede a la búsqueda de las propiedades relacionadas con el tipo de elemento actualmente en proceso, obteniendo solo las que necesite, para posteriormente obtener los valores de esas propiedades y aplicarlas al elemento. En el caso del que valor no tenga el formato correcto o no exista, se aplicará al elemento un valor por defecto para su estilo.

El siguiente fragmento es el encargado de obtener el valor de propiedad correspondiente para un elemento (por ejemplo, obtener el color una figura) y aplicarlo:

```
1. var propsCss = parseProperties( "shapes", "."+classname );
2. propsForShapes[ "color" ] = parseColor( propsCss[ "color" ] );
3. …
4. var shape = getShape( json, propsForShapes );
```

Donde:

1. Obtenemos las propiedades necesarias para las figuras
2. Se parsea la propiedad, devolviendo el valor correspondiente si es correcto, o el valor por defecto en caso contrario.
3. Otras instrucciones complementarias, como el parseo de otras propiedades.
4. Se crea la figura a partir del json y las propiedades de su estilo.

Cabe decir que esta metodología para obtener estilos puede ser aplicada también en futuros tipos de representaciones, aprovechando las funciones desarrolladas y que están disponibles en el script ***`main/vz-css-parser.js`***.

4.2 Anidamiento

Una estructura de datos es una buena forma de organizar y manejar grandes cantidades de datos, los cuales pueden ser simples (un carácter, un número...) o compuestos, que están formados a su vez por otros datos simples y/o compuestos (por ejemplo, un texto, formado por varios caracteres). En el caso particular de *VizHaskell* estos datos compuestos se corresponden con los tipos de representación anteriormente descritos además de los árboles, en los que los datos simples pueden ser el texto de la celda de la tabla, el valor del nodo de un

árbol, etc.. Los datos compuestos pueden anidarse de distintas formas. Por ejemplo, un árbol cuyos nodos son, a su vez, otros árboles. Esta formación de los datos compuestos mediante otros tipos de datos compuestos es lo que conocemos como anidamiento, permitiendo infinidad de niveles entre ellos salvo el caso particular de las escenas, que no podrán contener otras representaciones al ser una estructura final.

A continuación, entraremos en detalle sobre las posibilidades de uso del anidamiento en cada tipo de representación así como el funcionamiento para poder ver las estructuras anidadas.

4.2.1 Visualización de los tipos anidados

Dado que es inviable la presentación de la representación gráfica de las estructuras anidadas directamente en el interior de la representación en la que están contenidas (pues pueden alcanzar un tamaño considerable) se decidió hacer uso de ventanas emergentes en las que se mostrarían dichas representaciones de una manera directa.

Anteriormente, el anidamiento de los árboles se realizaba mediante la creación de nuevos elementos en la salida de la consola, donde se dibujaban. Esto podría suponer una visualización confusa de los datos, pues se mezclaban elementos de los distintos niveles jerárquicos de la estructura, no quedando claras las relaciones entre ellos.

Para la creación de las ventanas emergentes se ha hecho uso de la librería de *Bootstrap*, *bootstrap3-dialog-js* (**Dante, 2013**), la cual ofrece diversas posibilidades como la construcción de los mismos de una forma dinámica, eliminando la sobrecarga el *DOM* dada la cantidad de estructuras anidadas que pueda tener una representación en la que cada uno necesitaría de su propio elemento *HTML*. También nos permitía mostrar la ventana emergente con tan solo aportar un número pequeño de datos, como son su título, su contenido, los botones[31]...

La función que crea la ventana emergente, haciendo uso de la librería mencionada, se ha incluido en el fichero ***`main/dialog.js`***, necesitando los parámetros descritos en el párrafo anterior. Dicha función, ***`openModal,`*** es llamada por los diversos scripts que generan las representaciones. Estas ventanas emergentes generalmente tendrán de título el del elemento que se ha

[31] Las ventanas emergentes de las representaciones gráficas no hace uso de botones, pero estos si necesarios en el uso de los formularios, descrito más adelante, en el apartado 4.3 Interactividad.

seleccionado, proporcionándole al usuario una retroalimentación que le permita saber en todo momento dónde está.

El siguiente fragmento de código es el encargado de mostrar una representación en una ventana emergente:

```
1. var content = $( '<div/>' ).attr( 'id', idRep )
2. openModal( title, content );
3. content.append( working );
4. window.setTimeout( function(){
5.  addRepresentation( idRep, d.value, csss, options )
6.  content.find( '#loading-rep' ).remove();
7. }, 500 );
```

Donde:

1. Se crea el elemento que irá en el cuerpo de la ventana, donde se mostrará la representación gráfica.
2. Se muestra la ventana, aportando el título y el contenido creado en el punto anterior.
3. Se muestra el icono de espera al usuario, indicando que hay un procesamiento en marcha.
4. Dado que la ventana demandada en la instrucción 2 tarda un pequeño tiempo en crearse y mostrarse, se espera un intervalo de medio segundo para añadir la representación[32].
5. Se construye la representación gráfica pertinente.
6. Se elimina el icono de espera una vez terminada la construcción.

Cabe destacar que no hay un máximo de ventanas emergentes que podamos tener abiertas a la vez, esto viene dado según el número de niveles de anidamiento existentes en una representación gráfica.

4.2.2 Posibilidades de anidamiento

En este apartado procederemos a explicar las posibilidades de anidamiento y las combinaciones entre los distintos tipos de representación.

[32] Esto es debido a la naturaleza de *jQuery*, en la que no se pueden trabajar con elementos aún inexistente, necesitadas por los scripts que construyen las representaciones gráficas.

4.2.2.1 Árboles

Los árboles permiten ser añadidos a su vez en tablas u otros árboles, además de permitir que en sus nodos se puedan añadir otros árboles, tablas y escenas.

Para anidar estructuras en los árboles, estas se añadirán en el objeto *JSON* que representa el nodo de los árboles. Cuando se quieren acceder a ella, basta con pulsar sobre dicho nodo.

4.2.2.2 Tablas

Las tablas permiten ser añadidas a su vez en árboles u otras tablas, además de permitir que en cada una de sus celdas se pueden añadir tablas, árboles y escenas.

Para proceder a su anidamiento de estructuras en las tablas, se añadirá el objeto *JSON* que representan estas estructuras directamente sobre los objetos que simbolizan las celdas, es decir, en cada elemento del array **`values`** que representa los valores de la fila de la tabla.

Para poder acceder a las estructuras anidadas, las celdas de este tipo disponen de un botón, *Ver más*, cuyo comportamiento es abrir la ventana en la que se pintará la representación gráfica.

4.2.2.3 Escenas

Las escenas no admiten anidamiento ninguno, ya que se trata de un tipo final, pero sí que puede ser anidado en los demás tipos de representación, según el mecanismo (explicado anteriormente) de cada uno de ellos.

> *Véase anexo IV en el que se muestra un ejemplo sencillo de una representación gráfica anidada a otra con su correspondiente JSON.*

4.3 Interactividad

Terminados los nuevos tipos de representación, llegamos al objetivo más importante del proyecto: la interacción con las visualizaciones.
El aplicar la interactividad a los tipos de representación disponibles en *VizHaskell* ofrece al usuario la posibilidad de poder crear o modificar estructuras de datos fácilmente mediante el ratón e incluso modificar y eliminar datos de las mismas, suprimiendo el trabajo de escribir tediosas cadenas de expresiones, dada la naturaleza de *Haskell*.

La interactividad normalmente se hace más efectiva con el uso de interfaces gráficas, por lo que fue nuestra decisión del uso de las mismas para conseguir el objetivo. En este caso, se usan menús contextuales, que son elementos gráficos que de una forma efectiva, sencilla y ordenada muestran información y una serie de opciones potencialmente utilizables por el usuario y que internamente tienen implementada la acción requerida. Completando la función de los menús, se hace uso de formularios donde el usuario especifica los parámetros de construcción y modificación que afectan a las estructuras de datos.

4.3.1 Contrato JSON

Para los menús contextuales usamos esta estructura *JSON* general, la cual es común a todos los tipos de representación:

```
"commands": [
   "caption":"",
   "iconClass":"",
   "command":"",
   "parameters": [
               {
                      "id":"",
                      "caption",
                      "values": ["", "",..., ""],
                      "required":""
               }
   ],
   "subcommands": [
               {
                      "caption":"",
                      "iconClass":"",
                      "command":"Command with parameters",
                      "parameters": [
                             {
                                    "id":"",
                                    "caption":"",
                                    "required":""
                             }
                      ]
               },
               {
                      "id":"",
                      "caption":"",
                      "command":"Command without parameters"
               }
   ]
]
```

Donde:

`commands:` Array opcional que representa los menús de los elementos de los tipos de representación, en el que se puede incluir los siguientes campos:

- **`caption:`** Campo obligatorio en el cual se indica el texto a mostrar en la opción del menú, que además será el título de la ventana modal asociada a esa opción (si se diera el caso en el que el comando lleva parámetros).
- **`iconClass:`** Campo opcional que permite colocar un icono junto al título de la opción.
- **`parameters.`** Campo obligatorio que consta de un array con los parámetros requeridos (u opcionales) para la formación del comando. Si este array se encuentra vacío el comando se enviará directamente al servidor. Por el contrario, se abrirá un formulario en un cuadro de diálogo para poder incluir los parámetros (los cuales serán de tipo *input* o *select*). Cada parámetro estará formado a su vez por los siguientes campos:
    - **`id.`** Campo obligatorio que es el identificador (o *marca*) utilizado para saber en qué parte del comando se debe incluir el valor introducido.
    - **`caption.`** Campo obligatorio en el que se encuentra la etiqueta que se mostrará para este parámetro dentro del formulario.
    - **`values.`** Campo opcional que consta de un array que, en el caso de existir y no esté vacío, contendrá los valores disponibles para este parámetro. Podemos utilizar esta opción para restringir sólo al uso de ciertos valores. En este caso, el control utilizado para solicitar información al usuario será de tipo *select*.
    - **`required.`** Booleano opcional el cual si toma el valor **`true,`** entonces el parámetro debe ser incluido en el formulario obligatoriamente, por lo que se desactivará el botón *Ejecutar comando* hasta que se proporcione un valor. Si por el contrario es **`false`** (o no existe este campo) el parámetro es opcional, por lo que sí enviamos el comando estando este parámetro vacío, no se tendrá en cuenta y se eliminará la *marca* de la cadena del comando.

- **`command:`** Campo obligatorio si no existe **`subcommands`** y no permitido en caso contrario, representando la cadena del comando a ejecutar. Si se ha incluido el campo **`parameters`**, los parámetros de dicho comando deben ser rellenados previamente en un formulario al seleccionar la opción en el menú. Por el contrario, se enviará directamente al servidor sin realizar ninguna modificación.

Los *identificadores* (o *marcas*) de los parámetros dentro de esta cadena de comando deberán comenzar obligatoriamente por el carácter **?**, seguido del ID del parámetro que se quiere sustituir.

- **`subcommands:`** Campo obligatorio si no existe **`commands`** y no permitido en caso contrario. Representa el submenú para el elemento **`caption`** del menú, que consta de un array con las opciones disponibles para ese submenú. Los campos son los mismos anteriormente descritos en **`commands: caption, iconClass, parameters, id, command`** y **`subcommands.`**

Véase anexo V en el que se muestra un ejemplo sencillo de una representación gráfica con su menú contextual con su correspondiente JSON.

4.3.2 Estructura

Como hemos mencionado anteriormente, hacemos uso de menús contextuales y formularios, por lo que a continuación procederemos con la descripción de los mismos.

4.3.2.1 Menús Contextuales

La estructura de los menús contextuales es la típica que podemos encontrar en cualquier otra aplicación o incluso en el entorno gráfico de un sistema operativo. Para su construcción se utilizaron los plugins de *jQuery*, siendo estos *jQuery-contextMenu* (**Brala, 2011**) para los menús contextuales como tal y *jQueryUI*[33] para su posicionamiento.

Dentro de una página web, un menú contextual es una ventana que se abre como respuesta al evento generado por la pulsación del botón derecho del *mouse,* compuesto por una lista de opciones donde cada una a su vez puede generar otro menú, esta vez pulsando en la opción o situándose en ella.

La siguiente instrucción crea un menú contextual y lo asocia a uno o varios elementos:

```
1. addContextMenu (prefix, elemId, jsonCommands, idGraph, repType, event);
```

[33] http://jqueryui.com/

Los elementos asociados al menú vienen dados por la unión de los dos primeros parámetros, con las que se forma un id (**#id**) o un nombre de clase (**.class**). Los siguientes parámetros son, respectivamente, el *JSON* requerido para la contrucción del menú, el id del elemento que contiene la representación gráfica, el *repType* de la representación gráfica y el evento producido al pulsar el botón derecho[34] del ratón.

Dado que *jQuery-contextMenu* permite la creación submenús, podemos llegar a tener tantos niveles como anidamientos existan entre el campo **`commands`** y los correspondientes **`subcommands`**, los cuales se crean usando la función ***`contextMenuActions`*** de forma recursiva, dentro del script ***`app/main/vs-context-menu.js`***.

4.3.2.2 Formularios

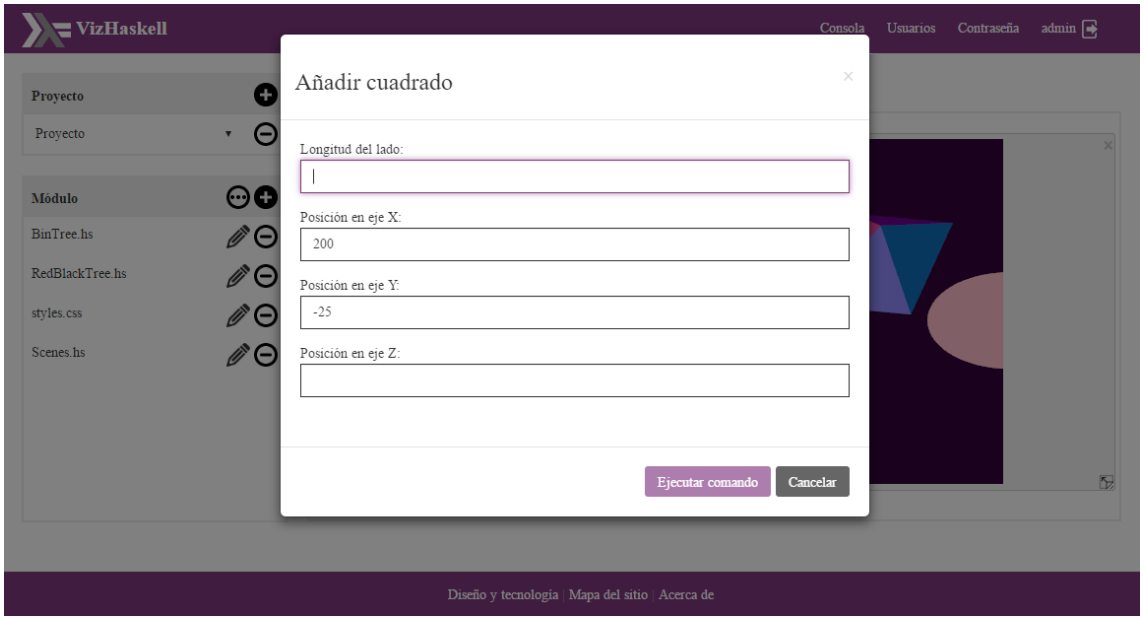
Los formularios sirven para recolectar datos introducidos por un usuario, que en nuestro caso son los valores de los parámetros que se pueden necesitar para la construcción de un comando y su posterior ejecución. Los elementos HTML del formulario se forman de manera dinámica utilizando *jQuery,* según los parámetros incluidos en el campo **`parameters`** del contrato *JSON,* entre los que distinguimos *input* o *select* con sus correspondientes *label*, indicado en el campo **`caption`**. La inclusión de uno u otro depende a su vez de la existencia del array **`values,`** colocando un *input* si este no existe, o por el contrario, colocando un *select* cuyas opciones son las del dicho array.

Los formularios se muestran mediante una ventana emergente de manera similar a como se hace en los anidamientos de representaciones descritos en los apartados anteriores del presente documento. Esta ventana, además de estar formado por el cuerpo del formulario, dispone de dos botones:

- **Cancelar.** Como el propio nombre indica, cancela la operación y cierra la ventana.

[34] Esto es usado por las escenas, las cuales son un caso especial que se explicará en el apartado 4.3.3 Funcionamiento.

- **Ejecutar comando.** Construye el comando a partir de los parámetros introducidos y lo envía el servidor para su ejecución. Cabe decir que estará desactivado, evitando el envío el comando, si en el formulario existe un campo obligatorio que se encuentre vacío, indicado por el campo **`required`**, manteniéndose de esta forma hasta que se le asigne un valor.



4.3.3 Funcionamiento

En este apartado explicaremos el algoritmo para la construcción de un comando, necesario para su ejecución posterior.

Mediante los menús contextuales podemos ejecutar tanto comandos sin parámetros como con ellos. Este consistirá en un *String* se obtenido del campo **`command`**. Los tipos de comandos son:

- **Sin parámetros.** En el caso en que no exista el campo **`parameters`**, el comando se enviará tal cual al servidor, sin ningún procesamiento previo[35].
- **Con parámetros.** En el caso en que sí exista el campo **`parameters`**. En esta situación, el comando, de nuevo extraído del campo **`command`**, necesita de un procesamiento previo para obtener los parámetros necesarios. Este procesamiento se explicará a continuación.

[35] Salvo el que se explicará en el *Apartado 4.4 Backreferences*.

4.3.3.1 Procesamiento de comandos con parámetros

Para procesar los comandos con parámetros, estos deben añadirse de alguna forma. Para ello, utilizamos unas *marcas* especiales, precedidas por el carácter **?**, dentro de la cadena de comando, las cuales se sustituirán por el valor del parámetro que introduce el usuario a través del formulario, es decir, mediante los *input* y *select* del mismo.

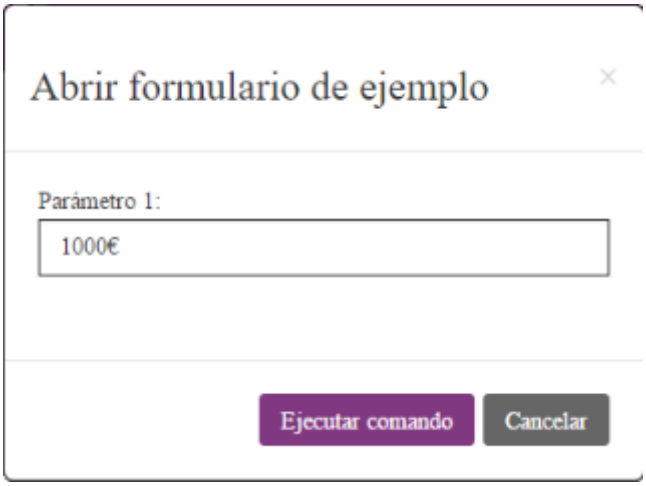
Estos *input* y *select*, tienen como *id* del *DOM* el mismo que viene reflejado en el campo **`id`** de cada parámetro, de tal forma que cuando se pulsa el botón *Enviar comando*, se recorren todos los elementos del formulario mediante *jQuery*, leyendo sus *id* de elemento y buscando ese *id* en la cadena de comando, para luego poder reemplazarlo por el valor que contenga ese elemento. Para una mayor claridad, se expone el siguiente ejemplo:

Supongamos la siguiente estructura *JSON:*

```
"commands": [
        "caption":"Abrir formulario de ejemplo"
        "command":"El valor del Parámetro 1 es: ?PARAM1",
        "parameters": [
                {
                        "id": "PARAM1",
                        "caption":"Parámetro 1",
                }
        ]
]
```

Esto generará un menú contextual con una sola opción , en el que al pulsar sobre esta se nos mostrará un formulario con un solo parámetro, es decir, un input *con su correspondiente* label *(“Parámetro 1”). Si se introduce un valor cualquiera, como por ejemplo, 1000€, y se pulsa* Ejecutar comando*, dado que coincide el valor del id del parámetro con la marca dentro del comando y esta viene precedida por* **?***, 1000€ se reemplazará en el comando. Como ya no hay más parámetros, el procesado del comando termina y ya comienza su ejecución.*

Antes: `"El valor del Parámetro 1 es: ?PARAM1"`
Después: `"El valor del Parámetro 1 es: 1000€"`



Este sería el comportamiento idóneo y esperado para los formularios pero hay ciertos casos a tener en cuenta, en los que a pesar de seguir siendo correctos y no generar errores, el reemplazamiento no se lleva a cabo. Los casos son los siguientes:

- **El comando requiere parámetros pero no existen las *marcas* en su cadena.** Como no existen las marcas en el comando, aunque introduzcamos los valores necesarios en el formulario, no se reemplazará nada, dejando la cadena sin modificarse.
- **El comando no requiere ningún parámetro pero existen *marcas* en la cadena.** Como vimos en el caso de comandos sin parámetros, el comando se enviará sin procesamiento previo alguno. Posiblemente puede provocar que la expresión Haskell esté mal formada, generando así un error en el intérprete.
- **El comando requiere un número *x* de parámetros pero hay más de *x* marcas en la cadena.** En este caso, las marcas no relacionadas con ningún parámetro serán eliminadas de la cadena, siempre que venga precedido por el carácter **?**.
- **El comando requiere parámetros opcionales, pero no se han introducido en el formulario, a pesar de existir *marcas* en la cadena.** En este caso, simplemente se eliminará la *marca* de la cadena del comando. Esto puede provocar que la expresión Haskell esté mal formada, generando así un error en el intérprete.

Una vez reemplazados (o no), todas las marcas por su valor de parámetro, el comando se escribe automáticamente en la entrada de la consola y se envía al servidor para su ejecución.

4.3.3.2 Menús contextuales en los tipos de representación.

En este apartado se explicarán los menús contextuales para cada representación gráfica.

4.3.3.2.1 Árboles

Los elementos de los árboles que tendrán la capacidad de mostrar los menús contextuales son los nodos. El menú asociado a cada nodo se configura durante la construcción del árbol, es decir, se asocia a su elemento *HTML*, utilizando para ello el objeto JSON adjunto a la estructura de dicho nodo.

En conclusión, cada nodo podrá tener o no, su propio menú contextual.

4.3.3.2.2 Tablas

Los menús contextuales irán asociados al elemento que permita su visualización, y se configuran durante la creación del mismo utilizando para ello el objeto *JSON* adjunto a la estructura de cada uno de ellos.

Los elementos de las tablas que tendrán la capacidad de mostrar los menús contextuales son los siguientes:

- **Celdas del cuerpo de la tabla.** El menú contextual se construirá a partir del correspondiente **`commands`** de la estructura de la celda. Puesto que trabajamos con tablas asociativas se podrán incluir menús a todas celdas, salvo a las que componen las claves de la tabla.
- **Indicadores de las filas.** El menú contextual se construirá a partir del correspondiente **`commands`** de la estructura de la fila, el cual afectará a toda ella.
- **Título de las columnas.** El menú contextual se construirá a partir del correspondiente **`commands`** de la estructura de la columna, el cual afectará a toda ella.

4.3.3.2.3 Escenas

Los elementos de las escenas que tendrán la capacidad de mostrar los menús contextuales son las figuras y la escena en general. El menú asociado a cada una de ellas se configura de forma dinámica según los elementos sobre los que se pulsa con el botón derecho.

Dado que el canvas de las escenas no extiende al *DOM*, no podemos tener un comportamiento predefinido para sus distintos elementos, lo que hace necesario

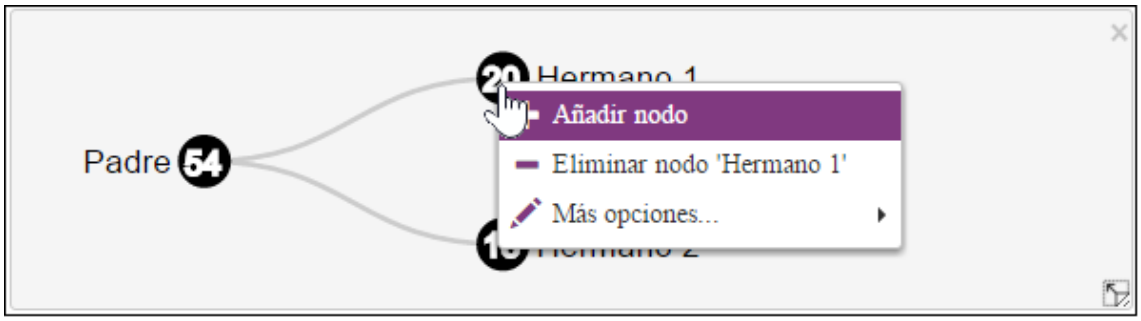
que estos se creen de forma dinámica mediante algún mecanismo. Para poder realizar esto, se vio necesario la utilización de *RayCaster* cuyo principio básico es proyectar un rayo que incide desde el punto en el que se pulsó y así obtener el primer objeto con el que colisiona.

El fragmento de código encargado proyectar el rayo y mostrar el menú correspondiente es el siguiente:

```
1. var raycaster = new THREE.Raycaster();
2. …
3. var intersects = raycaster.intersectObjects( sceneObjects );
4. var ojbCmds = intersects[firstShape].object.commands;
5. addContextMenu( '#', canvasId, ojbCmds, canvasId,"scene", event );
```

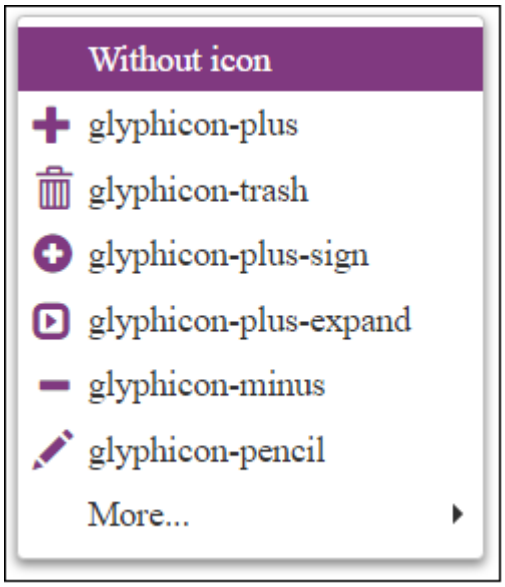
Donde:

1. Se crea el objeto RayCaster
2. Se configura el Raycaster mediante instrucciones complementarias no mostradas aquí.
3. Devuelve los objetos con los que ha colisionado.
4. Obtenemos la primera figura con la que intersecciona el rayo proyectado.
5. Se crea y muestra el menú generado a partir de los comandos asociados al elemento obtenido. Si no se ha detectado ningún elemento se mostrará el menú de la escena.



4.3.4 Estilo

En cuanto al estilo de los menús contextuales se puede añadir, mediante la estructura de los comandos, un icono para cada opción. Para que estos se visualicen correctamente, se deberá utilizar el campo **`iconClass`** cuyo valor tiene que comenzar por la palabra *glyphicon* seguido del nombre del icono elegido, pudiendo utilizar cualquiera de los disponibles en *Bootstrap*[36].



En un futuro, se pueden incluir en el proyecto más paquetes o librería de iconos. El color empleado para los menús es el característico de *VizHaskell* y tanto su diseño como el de los formularios están basados en las pautas de estilo de *Bootstrap.*

4.4 Backreferences

En el contexto de las expresiones regulares, los backreferences, o *referencias hacía atrás*, son mecanismos de programación que nos permiten hacer referencias a una subexpresión, la cual se encuentra entre paréntesis, desde otro punto de la expresión en la que está contenida.

En nuestro caso, hemos desarrollado dos funcionalidades basadas en este mecanismo pero ligeramente distintas, la cual afecta a las expresiones de Haskell. En lugar de hacer referencias a subexpresiones contenidas en una expresión, estas se realizan directamente a otras expresiones previamente evaluadas, de manera que podemos reemplazar parte de una expresión por otra expresión completa previamente evaluada. La funcionalidad de los

[36] http://getbootstrap.com/components/

backreferences según su contexto son distintas las cuales describiremos a continuación.

4.4.1 Referencias al historial de expresiones

Este tipo de *backreference* se basa en el uso de referencias a expresiones anteriormente evaluadas en la consola usando para ello una historia. Sus contextos son tanto la consola como las acciones de las opciones de los menús contextuales, pudiendo disponer de su uso en ambos ambientes.

4.4.1.1 Historial

Para conseguir este funcionamiento se ha tenido que implementar un mecanismo para poder ir almacenando los comandos ejecutados, es decir, crear un historial. Por ello, debimos meternos en las entrañas de la consola, en concreto, en el componente *Wterm,* el cual simula dicha consola. *Wterm*, a pesar de disponer ya de un historial de comandos, utilizado en el input de la consola al pulsar los botones *flecha arriba* y *abajo* del teclado para poder recorrerlo, no servía para nuestro propósito, pues solo almacena los últimos 100 comandos ejecutados durante la sesión, incluyendo aquellos comandos incorrectos en su expresión, haciendo necesario la implementación de un nuevo historial, independiente del otro.

Los pasos realizados en el script ***`extmod/wterm.jquery.js`*** fueron los siguientes:

1. Declaración del array, inicialmente vacío, que representaría el historial (**líneas 151** y **152**).
2. Creación de las funciones tanto para insertar un nuevo comando (**línea 283**) en el historial como para obtenerlo según su índice (**línea 296**). Si a la hora de obtener un comando el índice se encuentra fuera de rango, se devolverá una cadena vacía. Estas funciones extienden a *Wterm*, de manera que pueden ser utilizadas externamente de forma pública.

Hay ciertas situaciones en las que el comando ejecutado no se añadirá al historial. Dicho de otra forma, se añadirá una cadena vacía. Las situaciones son las siguientes:

- Cuando hemos introducido un *alias* incorrecto o no contemplado por la aplicación. Por ejemplo, **:mkoc** en lugar de **:mock**.
- Cuando el resultado devuelto por el servidor es un mensaje de error generado por el intérprete.

4.4.1.2 Funcionamiento y desarrollo

Ya creadas las funciones necesarias del historial, pudimos empezar con el desarrollo de las *backreferences*. Estas funcionan de la siguiente manera: antes del envío de una expresión hacia el servidor, debe realizarse un procesamiento previo, en el cual si en alguna parte de la cadena de comando se ha incluido la expresión **`@x`**, siendo **`x`** un número natural, dicha expresión será sustituida por el comando que obtengamos al acceder al índice **`x`** en el array del historial de comandos[37]. Para una mayor claridad, se expone el siguiente ejemplo:

> *Suponiendo que acabamos de iniciar una sesión con nuestro usuario y aún no se ha ejecutado ningún comando en la consola, realizamos los siguientes pasos:*

1. *Introducimos y ejecutamos en la consola la expresión* **`2+3`***, dando como resultado el valor* **`5`***. El comando se almacenará en la primera posición del array.*
2. *Introducimos y ejecutamos[38] en la consola la expresión* **`7-4`***, dando como resultado el valor* **`3`***. El comando se almacenará en la segunda posición del array.*
3. *Introducimos y ejecutamos en la consola la expresión* **`@1+@2`***. Llegados a este punto, como el procesamiento ha encontrado dos expresiones del tipo* **`@`****, procede con el reemplazamiento de los mismos por los comandos correspondientes. Respectivamente, sustituye las expresiones por el primer y segundo comando del historial, es decir, por* **`2+3`** y **`7-4`***, dando como resultado la expresión* **`2+3+7-4`** *que se enviará al servidor para su evaluación, devolviendo el resultado* **`8`***.*

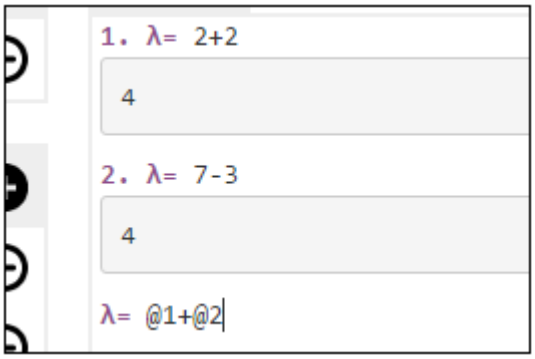
Esta funcionalidad también se aplicará sobre los comandos generados por los menús contextuales explicados anteriormente, puesto que estos insertan de forma automática el comando en el input de la consola, y es en el momento justo al envío del servidor cuando se realiza este procedimiento. Esto aporta una gran ventaja en la interactividad, pues podemos crear complejas expresiones de forma automática mediante este método, facilitando la modificación de las representaciones gráficas.

[37] El modo de implementar esto recuerda hasta cierto punto al funcionamiento de las marcas explicadas en el apartado de interactividad.

[38] Tanto esta expresión como la anterior han pasado por el procesamiento pero no han sido alterados de ninguna forma ya que no hay nada del tipo **@x**.

4.4.1.3 Feedback

Podemos pensar que un usuario, tras la ejecución de muchas expresiones en la consola, puede llegar a perderse al no saber cuántas ha ejecutado y en qué momento, confundiéndose a la hora de utilizar referencias de la forma `@x`. Como consecuencia de este tipo de situación, nos vimos en la necesidad de poder evitarla de alguna manera. La forma ideada fue la de incluir, junto al prompt de las expresiones en los resultados de la consola, un número que indicará el momento u orden en el que se ejecutó dicho comando.



4.4.2 Referencia a la expresión origen

Este segundo tipo de *backreference* se utiliza en aquellas expresiones en la que se quiere hacer referencia a la expresión original de la que resulta. Su contexto son únicamente las opciones de los menús contextuales en las representaciones gráficas, generalmente para referenciar la expresión responsable de esa representación, independientemente de su posición en el historial de expresiones evaluadas. Esto es necesario ya que el intérprete en el servidor necesita de esa expresión para crear de nuevo la representación con los cambios ya realizados en su estructura y como no tiene acceso al historial del cliente, no se puede hacer uso del otro tipo de *backreference*, necesitando por ello enviar la expresión final ya lista para su evaluación.

4.4.2.1 Funcionamiento y desarrollo

Su funcionamiento es simple, y no se ha de disponer de ningún tipo de historial. En este caso, si en una expresión generada por los menús contextuales, o sus correspondientes formularios, encontramos una cadena del tipo @@, se procede con la sustitución de dicha cadena por la expresión origen de la que resulta la representación gráfica. Antes de realizarse dicha sustitución, la expresión origen debe ser procesada para obtener solo la parte que nos interesa, que es la que se refiere a la expresión a representar encontrándose normalmente en la parte

final de la cadena. Este procesamiento tiene en cuenta tanto paréntesis como posibles corchetes de arrays y comillas para cadenas o caracteres. Para una mayor claridad, se expone el siguiente ejemplo:

1. *En la consola hemos enviado la expresión* **`putStrLn $ render RepresentationString (insert "valor 1" arbol3)`**, *dibujándose como resultado un árbol.*
2. *En el árbol resultante mostramos el menú de alguno de sus nodos, y elegimos la opción* Añadir un nodo*, cuya expresión almacenada es* **`putStrLn $ render RepresentationString (insert "?MARC1" @@)`**.
3. *Tras rellenar el formulario con el valor del nodo que se desee (por ejemplo,* **`valor 5`***), la expresión después del procesamiento de la interactividad será* **`putStrLn $ render RepresentationString(insert "valor 5" @@)`**.
4. *Al pulsar el botón* Ejecutar comando*, se iniciará en primer lugar el procesamiento para las cadenas del tipo @@, en la que se limpiará la expresión origen y se sustituirá en el sitio correspondiente dentro de expresión de la opción del menú, dando como resultado la expresión final. En segundo lugar, se realiza el procesamiento para las cadenas del tipo @***`X`***, que en este caso concreto no hay ninguna. Ya lista para su evaluación, la expresión final será* **`putStrLn $ render RepresentationString (insert "valor 5" (insert "valor 1" arbol3"))`**.

Hay que tener en cuenta que la expresión origen está libre de errores sintácticos ya que si se ha llegado a construir la representación gráfica, es porque precisamente esa expresión era correcta. Por el contrario, en la expresión final no se garantiza que no haya errores tras las sustituciones.

4.5 Casos de pruebas y ejemplo: los *Mocks*

En programación, los *mocks* son objetos de prueba que permiten imitar o simular el comportamiento de un objeto real ante una situación. Son utilizados generalmente para la realización de pruebas o cuando no se dispone del objeto final en el desarrollo de una funcionalidad. En nuestro caso, los *mocks* han servido para poder realizar el desarrollo de los tipos de representación en paralelo al desarrollo de los módulos de *Haskell* por parte de nuestro director de proyecto. Estos *mocks* simulan los objetos generados como resultado de la ejecución de esos módulos *Haskell*, es decir, representa la estructura *JSON* de los distintos tipos de representación, necesarios para el procesado y

construcción mediante los *script* de la aplicación. También nos ha servido para crear diversos ejemplos de prueba.

Para llevar a cabo el desarrollo de esta funcionalidad, más bien orientada a los desarrolladores, hemos tenido que introducirnos tanto en la parte del cliente como en el archivo *PHP* del servidor, utilizando los denominados *servicios*.

4.5.1 Implementación en el cliente

Para la parte del cliente, ha sido necesario crear un *servicio* que se apoye en AngularJS para comunicarse con el servidor, que es quién proporciona el *mock* solicitado. Para ello, ha sido necesario la realización de una serie de pasos:

1. Se ha creado el *servicio* en el script correspondiente, es decir, en la **línea 54** de ***app/models/model.js***, Se encarga de inyectar en la URL de la petición tanto el recurso solicitado (mock) como su id.
2. Se ha añadido el servicio al controlador correspondiente, en concreto, en la **línea 3** del fichero ***app/control/projects.js***.
3. Se ha creado la función en el controlador que hace uso del servicio (**línea 465** del fichero ***app/control/projects.js***).
4. Se ha creado la función **`ajaxCommandMock`** (**línea 53** de ***app/main/console.js***.) donde se configura las funciones a ejecutar en caso de éxito o error, de manera análoga a **`ajaxCommand`**,, además se hace uso de la función del controlador creada anteriormente.
5. Por último, también sobre el fichero ***app/main/console.js***, se ha añadido la cadena *mock* al filtro de comandos disponibles (**línea 188**) y la función a ejecutar en caso de introducir dicho comando en la consola (**línea 272**).

Dado que el uso de los *mocks* y el *servicio* correspondiente están orientados a desarrolladores, cuando la aplicación esté en producción es necesario que todo el código creado en los puntos anteriores no esté disponible. Para ello, se recomiendan dos alternativas:

- Eliminar las líneas de código desarrolladas. La opción más sencilla y rápida.
- Restringir el uso del comando **:mock** a ciertos usuarios, como son los administradores. La opción más compleja, dada la necesidad de desarrollar el código necesario para ello, pero también la ideal.

4.5.2 Implementación en el servidor

En esta parte, se ha tenido que modificar el fichero ***php/server.php*** que implementa la *API REST*. En concreto, se ha activado la opción de tratar las peticiones enviadas mediante el método *GET (***líneas 62 y 63***)*, donde se ejecuta una función en la que se procesa la *URL* para obtener el identificador del *mock* y devolver el ejemplo correspondiente, con su *CSS*, si procede. Estas respuestas devueltas siguen el mismo formato que las que se genera cuando se procesa un comando mediante el intérprete, es decir, un objeto en el que se incluye la estructura *JSON* y su posible texto *CSS*.

De manera similar a la parte cliente, cuando la aplicación esté en producción, el método *GET* no debe estar disponible en el caso que no se haya restringido su uso a los administradores, como originalmente se hacía antes de la realización de esta funcionalidad. Para ello, sería necesario eliminar las **líneas 62 y 63** del archivo ***php/server.php***.

4.5.3 Almacenamiento de los *mocks*

Los *mocks* se almacenan en un directorio de igual nombre, en ***www***. Dentro de este directorio están los archivos *.json* que los representan y tienen como nombre cualquier cadena válida, decidida por el creador del ejemplo. Acompañando a estos archivos de extensión *.json*, es posible que se encuentren opcionalmente los archivos de los estilos, con extensión *.css* y nombre el mismo de aquél con el que se relaciona.

Por otra parte, también ha sido necesario modificar el archivo ***php/phprest.ini***. En concreto se ha añadido en la **línea 6** el alias correspondiente a la ruta del directorio donde se almacenan los *mocks*.

4.5.4 Utilización de los *mocks*

Para poder hacer uso de los *mocks* desde la consola de la aplicación, basta con introducir el comando **`:mock`** seguido del id del que se quiere procesar. Por ejemplo, **`:mock scene1`**.

Por otro lado, tanto la mala utilización de este comando como disponer de un *mock* mal construido, puede llevar a la generación de errores, entre los que se encuentran:

1. Introducir el comando sin el id correspondiente. Ejemplo: **`:mock`**.
2. Introducir el comando con varios *id*. Ejemplo: **`:mock scene1 table2`**.

3. Solicitar un *mock* inexistente. Ejemplo: **`:mock mockInexsistente`**
4. Solicitar un *mock* cuya estructura JSON tiene errores.

GHC 7.6.3

```
1. λ= mock
ERROR MOCK: NO SUCH IDENTIFIER
2. λ= mock scene1 table2
ERROR MOCK: TOO MANY PARAMS
3. λ= mock mockInexistente
ERROR Invalid param: Mock 'mockInexistente' not found.
4. λ= mock jsonIncorrecto
ERROR Invalid param: JSON format for 'jsonIncorrecto' is incorrect.
Please, contact with the Webmaster.
λ=
```

Capítulo 5

Extensibilidad

En este capítulo procederemos a describir las diversas formas con la que podemos mejorar y extender las diferentes funcionalidades y componentes con las que hemos trabajado. Mencionaremos principalmente las relacionadas con nuestras contribuciones, puesto que muchas ideas ya han sido descritas en la memoria del proyecto anterior.

5.1 Mejorar las tablas

Las tablas, a pesar de tener un buen diseño con algunos elementos interactivos, pueden ser mejoradas con nuevos elementos visuales además de funcionales, como los que describiremos en los siguientes puntos. Como información útil, es opcional pero recomendable tener algunos conocimientos en el uso de *jQuery* para las elaboraciones de estas mejoras.

- **Navegación por las columnas.** De igual forma igual que ya se hace con las filas, podemos tener un mecanismo para ir recorriendo las columnas poco a poco. Esto puede ser efectivo para aquellas tablas común gran número de columnas. Los pasos que se pueden seguir para añadir esta mejora son:

    1. Añadir a la tabla los elementos requeridos, como un botón, haciendo uso de *jQuery.*
    2. Añadir un manejador de eventos a estos elementos. Este ejecutará un algoritmo similar al que se usa en las filas.

- **Opciones en la visualización de la tabla.** Actualmente solo es posible visualizar cinco filas a la vez, haciendo necesario el uso de los botones de navegación para recorrer toda la tabla. Para aportar cierta libertad y comodidad al usuario, puede ser ideal que este sea el que decida cuántas

filas a vez quiere que se muestren en cada momento. Por ello, podemos seguir los siguientes pasos:

1. Localizar el segmento de código donde se crean los elementos de la tabla, esto es, a partir de la **línea 60**, aproximadamente.
2. Crear elemento, como puede ser un *input*, situado entre los botones de navegación, por recomendar un lugar de la tabla, y que admita únicamente valores numéricos.
3. Añadir el manejador al elemento, por ejemplo, mediante la función `.on("submit")` de jQuery. El mismo debería ejecutar unas instrucciones que se encarguen de realizar esta nueva configuración.

- **Ordenación del contenido de la tabla.** Dado que en las tablas no se pueden interactuar más allá de su recorrido, se podría permitir una forma de ordenar su contenido, por ejemplo, de manera alfabética. Esta ordenación en principio solo estará permitida en las columnas que contiene valores de tipo *String*. Podemos seguir de forma orientativa los siguientes pasos:

  1. Durante la construcción de las columnas, obtener el tipo de datos que contendrán las celdas de la misma.
  2. Si el tipo obtenido es *text*, se procedería a añadir un manejador de eventos a la celda del título cuya funcionalidad es la de ordenar las filas a partir de esos valores.

5.2 Extender la funcionalidad de THREE

Es posible que las mejoras y funcionalidades que se añadan a las escenas hagan uso de un módulo en *THREE* que no esté no incluido actualmente en la aplicación. Si es así, hay que realizar los siguientes pasos:

1. Buscar el módulo necesario en la página oficial de la librería[39] (o en otras páginas externas) y descargarlo.
2. Introducir el módulo descargado en el directorio correspondiente de `www` y cambiar su propietario y sus permisos por `www-data` y `755`, respectivamente.

Agregar la ruta del módulo en el HTML principal de la aplicación, es decir, en `dviz.html` dentro de la sección donde se incluyen los ficheros JavaScript de la aplicación. Para un correcto funcionamiento, es necesario que se incluya debajo de otros módulos de los que pueda depender, siendo `three.js` uno de ellos.

[39] http://threejs.org/

5.3 Mejorar las escenas 3D

Las escenas en *VizHaskell* no están ni mucho menos terminadas ni exprimidas al máximo, pudiendo añadir a ellas una gran cantidad de mejoras como las que describiremos a continuación. Cabe destacar que para poder realizarlas seguramente se tenga que disponer de cierta experiencia en el uso de la librería *THREE*.

- **Añadir y permitir la elección de otros renderizadores.** Actualmente, las escenas usan *WegGL* por defecto en la renderización de las escenas, y *canvas 2D* si *WebGL* no está disponible. Se podrían añadir nuevos renderizadores además de dar al usuario la opción de poder escoger el que desee utilizar en sus escenas. Para ello y de una manera sencilla, podemos seguir los siguientes pasos:

    1. Incluir un nuevo campo, como puede ser **`render`**, en la raíz del contrato *JSON* de las escenas. Este campo debería incluir las opciones mínimas que pueda necesitar el renderizador en base al tipo que se escoja.
    2. Modificar la sección de código donde se crea el renderizador, al inicio de la función **`init()`**, donde presumiblemente será necesario leer desde el *JSON* el tipo de render y sus opciones.
    3. Posiblemente ya no será necesario el *detector* al inicio de la función **`addScene`**, pudiendolo mover y adaptar al código desarrollado en el punto anterior.

- **Añadir y permitir la elección de distintos tipos de cámaras**. Actualmente las escenas solo permiten el uso de cámaras cuya proyección se basa en perspectiva cónica y sus propiedades vienen predefinidas en el script **`/main/scene.js`**. Sería ideal añadir nuevos tipos de cámara, como la que usan una proyección basada en la perspectiva isométrica, además de permitir al usuario elegir entre el uso de una u otra. Para llevar a cabo esto, se pueden realizar los siguientes pasos:

    1. Incluir un nuevo campo, como puede ser **`camera`**, en la raíz del contrato *JSON* de las escenas. Este campo debería incluir las propiedades mínimas para crear la cámara, como su posición en la escena, su rotación…
    2. Modificar la sección de código donde se crea la cámara, en la función **`init()`** del script **`/main/scene.js`**. Es esperable que se haga una lectura del tipo y las propiedades desde el campo

correspondiente del JSON, además de la inserción de un IF-THEN-ELSE donde distinguir el tipo de cámara a crear.

- **Añadir una nueva figura.** Las figuras disponibles en las escenas de VizHaskell son muy diversas, pero podemos aun así añadir más tipos que potencialmente puedan ser usados por los usuarios. Para llevar a cabo esto, se pueden realizar los siguientes pasos:

    1. Diseñar la estructura *JSON* de la figura con los campos que pueda necesitar, además de los obligatorios, de forma similar a los que ya existen.
    2. Incluir un nuevo caso en el **`switch`** de la función **`getShape`** del script **`/main/scene.js`** con el nombre del nuevo tipo de figura. Dentro de este caso se debería llamar a la función que crea la geometría de la figura de la forma que el desarrollador estime oportuna.

    Si la figura se trata de una figura plana y queremos que este también pueda estar disponible como base de un prisma o una pirámide, hay que realizar además el siguiente paso:

    3. Incluir un nuevo caso en el **`switch`** de la función **`createPrism`** (o **`createPyramid`**) del mismo script, con el nombre de la base igual al de la nueva figura plana. Dentro de este caso se debería incluir el código para crear el prisma (o la pirámide) a partir de dicha base, de manera similar a los demás casos del **`switch`**.

- **Añadir un nuevo tipo de luz.** Las escenas en VizHaskell no incluyen todas las luces disponibles en *THREE*, por lo que también se pueden añadir más tipos de las mismas. Para realizarlo se pueden seguir los siguientes pasos:

    1. Diseñar la estructura *JSON* de la luz con los diferentes campos que puede necesitar, además de los obligatorios que ya existen.
    2. Incluir un nuevo caso en el switch de la función **`getLight`** del script **`/main/scene.js`** con el nuevo nombre del tipo de luz. Dentro de este caso se debería llamar a la función que crea la luz de la forma que el desarrollador estime oportuna.

- **Incluir nuevas fuentes para los textos 3D.** Los textos 3D de las escenas usan únicamente la fuente *Helvetike* por lo que se podría permitir la elección de más fuentes en los textos 3D. Hay que tener en cuenta que *THREE* utiliza un formato de fuente especial, distinta de los habituales. En concreto, los ficheros de las fuentes deben seguir el formato *JSON*. Para la inserción de nuevas fuentes se pueden realizar los siguientes pasos:

1. Obtener la fuente *[nameFont].typeface.js* de la página oficial de THREE, de páginas externas o mediante la herramienta online *Facetype.js*[40], que convierte las fuentes al formato requerido.
2. Añadir el archivo al directorio **`/ext/three/fonts`** y cambiar su propietario y sus permisos por **`www-data`** y **`755`**, respectivamente.
3. Agregar la ruta del fichero en el HTML principal de la aplicación, es decir, en **`dviz.html`** dentro de la sección donde se incluyen las rutas de los ficheros *JavaScript*, de manera similar al que ya viene incluido.
4. Actualmente no viene implementada la elección de la fuente, por lo que podríamos hacerlo mediante el uso de propiedades CSS utilizando, por ejemplo, la propiedad **`font`** o **`font-family`** con valor el nombre de la fuente que se quiera incluir, además de estos pasos extras:

    a. Añadir la instrucción necesaria para obtener la propiedad en las funciones **`getShape`** y **`createShape`** del script **`/main/scene.js`**.
    b. Añadir el nuevo parámetro en la función que crea la geometría del texto en la función **`createText`** y de la forma exigida por *THREE*.

- **Añadir nuevas propiedades CSS para los distintos elementos de la escena.** Las propiedades CSS son la mejor forma de añadir estilos a las escenas. La inclusión de nuevas propiedades sacaría partido a la creatividad de los usuarios. Para llevar a cabo esta mejora, se puede realizar los siguientes pasos:
    1. En el fichero **`css.jss`** que extiende la sintaxis CSS de **`codemirror.js`** añadir los siguientes cambios.

        a. Añadir el nombre de la nueva propiedad en el array de propiedades conocidas (**línea 372** o **464**, según si la propiedad es estándar o no).
        b. Añadir el valor, si procede, al array de valores conocidos (**línea 501**).

    2. Añadir al array que permite filtrar las propiedades no admitidas en las escenas del fichero **`/main/scene.js`** el nombre de la nueva propiedad (**línea 32**).
    3. En la función **`parseProperties`** del script **`/main/scene.js`** añadir la propiedad al array adecuado según el/los tipo/s de elemento/s que puedan necesitar la propiedad.

[40] Herramienta online: http://gero3.github.io/facetype.js/
Página web del proyecto: https://github.com/gero3/facetype.js

4. Parsear la propiedad leída y añadir el valor por defecto si el formato no es el correcto o no se ha incluido, dentro de la función adecuada (**`createLight`**, **`createShape`**, etc..).
5. Hacer uso de la propiedad en la función que construye el elemento que pueda utilizar esa propiedad (**`getLight`**, **`getShape`**, etc..).

Si la propiedad se trata de un nuevo material, además hay que realizar el siguiente paso:

6. Añadir un nuevo caso con el nombre del material al **`switch`** de la función **`getShape`** con el fragmento de código necesario para crear el material.

5.4 Mejorar los backreferences

Los backreferences actualmente funcionan mediante la utilización del historial de expresiones y mediante las referencias a una expresión origen. Sería interesante implementar otro tipo de procesamiento para los *backreferences*, de forma que en una expresión tecleada se puedan incluir referencias a partes de la misma, es decir, a sus propias subexpresiones, permitiendo así la sustitución de esas referencias por esas partes. Para ellos, se podría hacer uso de un carácter no reservado en Haskell ni en la aplicación, como por ejemplo, la barra vertical "`|`". Los pasos serían los siguientes:

1. Localizar la función **`ajaxCommand`** que actualmente procesa el comando antes de su envío al servidor en el script **`console.js`**.
2. Elaborar el algoritmo que ejecuta esta funcionalidad.
3. Llamar a la función que realizaría este procesamiento sobre la cadena de la expresión, pudiendo incluir antes o después de la llamada del actual funcionamiento.

5.5 Añadir un tipo de presentación

Además de los tipos de representación existentes, se podría añadir uno nuevo, enriqueciendo en gran medida las posibilidades de la aplicación a la hora de mostrar estructuras de datos. Esta tarea puede requerir del aprendizaje de una nueva tecnología, si se diera el caso de no hacer uso de las librerías ya disponibles en VizHaskell. Los pasos generales pueden ser:

1. Elaborar el contrato *JSON* del nuevo tipo de representación.

2. Desarrollar el código fuente que haciendo uso del contrato construya la representación gráfica deseada.
3. En **`/main/console.js`** en la función **`responseProcess`** añadir un nuevo caso que a partir de su **`reptype`** que haga uso del código fuente desarrollado.

Capítulo 6

Conclusiones

Para finalizar con el presente documento, nos gustaría destacar los resultados obtenidos tras el trabajo realizado en la consecución de los objetivos marcados inicialmente.

Dado que nuestros objetivos eran cuatro, podemos decir que hemos logrado alcanzar el fin de una manera satisfactoria para cada uno de ellos, aunque también cabe mencionar que nos hemos encontrado con diversos problemas haciendo que los objetivos no sean tan completos como nos hubiese gustado.
Veremos en detalle qué objetivos hemos realizado y hasta qué punto hemos logrado llevarlo a cabo:

- **Incorporación de nuevos tipos de representación.** Para este objetivo teníamos interés en añadir dos tipos de representación, que son las tablas y escenas tridimensionales. A pesar de que siempre se puede mejorar o aumentar funcionalidad de las mismas, hemos conseguido realizar estas estructuras con los suficientes elementos que sirvan de utilidad para el usuario. En su elaboración el principal problema encontrado ha sido la necesidad de entrar en contacto con una nueva tecnología ajena a nosotros, por lo que nos ha supuesto un tiempo adicional en aprender y manejar las herramientas que fueran necesarias para diseñar y elaborar las estructuras.
- **Anidamiento.** Este objetivo, se podría decir que cumple nuestras expectativas debido a que todas las estructuras con las que hemos trabajado permiten tal fin. A pesar de tener que hacer uso de la estructura de datos que ya disponía la aplicación no nos ha supuesto un gran problema a la hora de poder relacionarlos entre sí.
- **Interactividad y manipulación de los tipos de datos.** Siendo este el más importante de los objetivos que planteamos, creemos que la funcionalidad es lo suficientemente robusta la hora de ser utilizada por los usuarios finales, pues añade una forma sencilla y completa para poder trabajar con sus estructuras de datos. La dificultad encontrada en su realización era la de pensar y encontrar los mecanismos adecuados para

cada tipo de representación gráfica existente, dada la diversidad de tecnologías utilizadas para cada uno de ellos.

- **Backreferences.** La idea principal era poder realizar varias funcionalidades relacionadas con el uso del historial, referencia a la expresión de la que resulta y referencias de subexpresiones de la misma. Como consecuencia del tiempo extra dedicado a solventar los objetivos anteriores, nos hemos encontrado con la adversidad de no llevar al completo este objetivo, en particular la última funcionalidad mencionada, de manera que decidimos explicarlo en el *Capítulo 5 Extensibilidad* para que en un futuro pueda ser terminado.
- **En general,** otra de las circunstancias que encontramos fue el esfuerzo extra para entender la arquitectura y el funcionamiento de la disponía la aplicación cuando ésta cayó en nuestras manos, puesto que era algo nuevo para nosotros.

A pesar de las dificultades encontradas, podemos decir con satisfacción que VizHaskell ha mejorado considerablemente ofreciendo nuevas funcionalidades al usuario al que está destinado, proporcionándole nuevas posibilidades como por ejemplo, mayor número de estructuras, la realización de operaciones con esas estructuras de manera sencilla además de poder visualizarlas e interactuar con ellas de manera gráfica y poder mostrar estructuras de datos más complejas. Son pocas las herramientas que ofrecen todas estas funcionalidades, por lo que hemos logrado obtener una aplicación que va más allá de ellas, pudiendo favorecer el entendimiento y aprendizaje de este lenguaje de una manera que sea más atractiva y original para captar la atención de nuevos usuarios que quieran sumergirse en el mundo de la programación funcional.

VizHaskell no solo supone una herramienta útil para los usuarios, sino que además ha sido algo más para nosotros durante estos duros y largos meses. Lo que nos ha aportado esta herramienta es poner en práctica muchos de los conocimientos adquiridos de la carrera, como por ejemplo, *JavaScript, php, html5, etc.* Incluso poder entender un poco más aquellos conceptos que no hemos tenido la oportunidad de estudiar, como es el caso de la programación funcional.

A parte de los aspectos técnicos aprendidos, nos ha aportado un gran crecimiento personal, muy bueno para nuestro inminente futuro en la vida laboral.

Trabajo futuro

Algunas de las mejoras que se pueden realizar en el caso de mejorar esta aplicación son:

- **Añadir vistas múltiples a las escenas (*viewports*).** THREE permite las escenas con múltiples vistas, o mejor dicho, varias cámaras, sin la necesidad de usar varios renderizadores para ello. Sería interesante añadir estas características para que los usuarios opcionalmente puedan potenciar la visualización de sus estructuras.
- **Añadir comportamientos en los elementos de las escenas 3D.** Los elementos de las escenas 3D no tienen por qué ser necesariamente estáticos, porque lo que se podrían implementar mecanismos que permitan animarlas.
- **Manipulación de las estructuras de datos en tiempo real.** Actualmente se envía una orden al intérprete en el servidor cada vez que se manipula una estructura. Se podría implementar un mecanismo para cada tipo de representación que permita modificar *en vivo* las representaciones. De esta forma, se guardarían localmente las expresiones (o se generaría una muy compleja) con todos los cambios a realizar en la estructura, para posteriormente enviarla al intérprete, y confirmar la modificación.
- **Añadir nuevos tipos de representación.** A pesar de los tres tipos de representación ya existentes, sería interesante la inclusión de nuevos tipos, haciéndose uso (o no) de las librerías que se disponen.

Conclusions

To finish with this document, we would like to highlight the obtained results from the work done in the achievement of the objectives initially set.

Given that our objectives were four, we can say that we have achieved each of them in a satisfactory way. However, we should also mention that we have encountered several issues which have made the objectives not as complete as we would have liked.
We will see in detail what goals we have done and how far we have managed to carry them out:

- **Incorporation of new types of representation.** For this purpose we were interested in adding two types of representation, namely tables and three-dimensional scenes. Although functionality can always be improved or increased, we have managed to develop these structures with a sufficient number of components that are useful to the user. The main problem we have encountered during its development is the need to make contact with new technologies beyond our control, which has involved additional time to learn and to use the necessary tools to design and build structures.
- **Nesting.** We can say that this objective has met our expectations, since all the structures we have worked with allow this purpose. Although we used an already implemented data structure, we had no problems with relating one another.
- **Interactivity and manipulation of data types.** Being the most important goal we have propose, we think that the functionality is robust enough for use by end users because it adds a simple and complete way to work with their data structures. The difficulty experienced in its implementation was thinking and finding appropriate mechanisms for each type of existing graphical representation, given the diversity of technologies used for each of them.
- **Backreferences.** The main idea was to develop several pieces of functionality related to the history usage, reference to expression from which it results and references to subexpressions thereof. Due to the extra time dedicated to solve the above objectives, we have met adversity in not

achieving our goal, in particular the last feature mentioned, so we decided to explain it in *Chapter 5 Extensibilidad* to be completed in the future.

- **In general terms,** another issue we have encountered was the extra effort to understand the architecture and functioning of the available features in the application when it fell in our hands, because it was something new for us.

Despite the difficulties encountered, we can say with satisfaction that VizHaskell has been greatly improved with new functionality to the users which is intended to. We have provided new possibilities: more types of structures, the application of some given operations to these structures in an easy way. Besides this, we can render, interact with, and display graphically complex data structures. There are few tools that offer these features, so we managed to get an application that goes beyond them and can foster understanding and learning of the Haskell language in a way that is more attractive and original, and captures the attention of new users who want to enter the world of functional programming.

VizHaskell not only is a useful tool for users, but has also been more for us during these hard and long months. What this tool has given us is to implement much of the knowledge acquired in the degree studies, such as *javascript*, *php*, *html5*, etc. We have been able to understand a few more concepts that we had not got the opportunity to study, such as functional programming.

Besides the technical aspects learned, it has given us a great personal growth, very good for our impending future working life.

Future Work

Some of the improvements that can be made to improve this application are:

- **Add multiple views to Scenes (*viewports*)**. THREE allows scenes with multiple views, or rather, several cameras, without the need to use multiple renderers for it. It would be interesting to add these features for users to improve the visualization of their structures.
- **Add behaviors to the elements of 3D scenes.** The elements of 3D scenes need not be necessarily static, so several mechanisms could be implemented to animate them.
- **Manipulation of data structures in real time.** Currently an order is sent to the interpreter in the server every time a structure is manipulated. It is possible to implement a mechanism for each type of representation that allows the programmer to change a structure in real time.

- **Add new types of representation.** Despite the three types of representation, it would be interesting to include new types, making use (or not) of the libraries that are available.

Capítulo 7

Aportaciones de los integrantes

En este reducido capítulo describiremos todas las actividades realizadas por los integrantes, sirviendo también para dar una idea global de todo el trabajo realizado. Cabe decir que a pesar de dividirnos el trabajo, hemos estado manteniendo una constante comunicación ya sea para asignarnos tareas o para solventar algunas dudas que pudieran surgir en el desarrollo de la misma, desde aspectos relacionados con el análisis y diseño a otros más técnicos como aportarnos ideas a la hora de implementar código. Incluso hemos trabajado de manera conjunta en una tarea específica si esta la requería. Comentaremos las actividades realizadas que van desde los inicios del proyecto con la instalación y configuración de la aplicación y las herramientas de desarrollo, hasta la elaboración de la memoria y la preparación del material de entrega. Inicialmente, para cada objetivo que debíamos realizar, procuramos separar las tareas de tal manera que cada uno tocara un poco de todo para que así entendiéramos el funcionamiento de todo el trabajo desarrollado una vez en el final del proyecto.

Lidia Flores

- **Preparado del entorno secundario de desarrollo y depuración mediante la instalación del material necesario.** Una vez instalada la máquina virtual, mientras David realizaba la configuración, me dediqué a buscar un escritorio adecuado para instalar en el Ubuntu del servidor, además de otras herramientas útiles y necesarias, como las *Guest Additions* o un editor de texto.
- **Elaboración del contrato JSON de las tablas asociativas.** Yo me dispuse a crear el contrato JSON para que nuestro director pudiera comenzar con los módulos en Haskell.

- **Implementación del algoritmo de navegación por las filas de la tabla.** Paralelamente a la implementación del algoritmo de búsqueda, yo me dediqué a implementar el algoritmo para la navegación por la tabla.
- **Elaboración de pruebas y depuración del script *table.js*.**
- **Desarrollo del código encargado de procesar las peticiones mocks en el lado de servidor (servicios REST de PHP).** De forma paralela a mi compañero, realicé el código necesario en el servidor para procesar las peticiones de *mocks* y la devolución del mismo.
- **Estudio, búsqueda y pruebas de librerías para la elaboración de un mecanismo para representar estructuras anidadas.** Dado que necesitamos de un medio para representar las estructuras anidadas, decidimos hacer uso de ventanas modales, por lo que me dispuse a investigar alguna librería útil para este propósito.
- **Modificación del script *table.js* para permitir anidamiento.** Añadimos a *table.js* el código necesario para permitir el anidamiento y su visualización en ventanas modales.
- **Estudio, búsqueda y pruebas de alguna librería de gráficos 3D.** Dado que íbamos a crear escenas tridimensionales, me dispuse a buscar alguna librería gráfica para la elaboración de este tipo de representación.
- **Elaboración de las funciones necesarias para elaborar la escena y las distintas figuras y luces.** A la par de la definición del contrato *JSON* a utilizar, me dispuse a realizar las funciones necesarias para elaborar las figuras escogidas sobre un *HTML* estático.
- **Inserción de un mecanismo de interacción en las escenas.** Dado que necesitábamos interactuar con las escenas para poder movernos libremente por ella, añadí el código en el lugar correspondiente haciendo uso de los mecanismos de los que dispone *THREE*.
- **Elaboración de las funciones necesarias en el script vs-parser-css para el parseo simple de textos CSS y su filtrado.** Dado que necesitábamos separar la estructura de las figuras y su estilo, debíamos disponer de un mecanismo para añadir esto último a los elementos mediante el CSS ya existente en la aplicación, por lo que me dediqué a buscar a librería adecuada para el parseo de CSS y su filtrado.
- **Incorporación y adaptación del código de las escenas en el archivo *scene.js* del proyecto.**
- **Elaboración de un contrato JSON general para todas las representaciones**. Dado que debíamos disponer de un mecanismo

común para hacer uso de los menús contextuales en todas las representaciones, elaboré la estructura *JSON*, apoyándome en mi compañero si era necesario.

- **Elaboración de los formularios en ventanas modales en base al contrato JSON.** Mientras se desarrollaba el código para los menús contextuales, realicé el código necesario para la construcción dinámica de los formularios que irían dentro de las ventanas modales solicitadas por los menús contextuales.
- **Construcción de una expresión en base a los formularios.** Dado que los valores del formulario pueden ser muy diversos, tenía que desarrollar un mecanismo para añadir esos parámetros a la expresión que se enviaría al servidor.
- **Incorporación de los menús contextuales en los scripts *table.js* y *tree.js.*** Para ello, debía realizar en ambos scripts el código necesario encargado de construir y llamar a los menús contextuales, mediante eventos generados por las acciones en los elementos HTML de los mismos.
- **Elaboración de una de las funcionalidades de los *backreferences*.** En concreto, yo me dediqué a la encargada de referenciar a expresiones del historial, mediante la realización del código necesario en los distintos *scripts* de la aplicación como *console.js, wterm.js o backreferences.js*.
- **Elaboración de algunos ejemplos de representaciones gráficas.**
- **Escritura de algunos apartados de la memoria.** Entre ellos están el índice y el capítulo de antecedentes, de las conclusiones y algunos apartados de contribuciones y extensibilidad.
- **Elaboración e inserción de las distintas imágenes de la memoria.**
- **Maquetado de la memoria mediante utilizando para ello Microsoft Word.**

David Bolaños

- **Instalación de la aplicación en Virtualbox y estudio y ajuste de la configuración necesaria.** Yo realicé la instalación de Ubuntu en VirtualBox, además de configurar el mismo para que se redireccionaran

los puertos entre ambas máquinas y añadir las líneas necesarias al archivo de configuración de *Apache* para el *proxy*.

- **Estudio y búsqueda de librerías útiles para utilizarse en las tablas.** Ya dispuesto y preparado el entorno de trabajo, me dediqué a buscar alguna tecnología que pudiera ser útil para la elaboración de las tablas. Dado que no encontré nada interesante, decidí junto a Lidia que su construcción se haría mediante jQuery.

- **Diseño, elaboración del HTML y maquetado con CSS de la tabla de forma estática.** Dado que no encontré ninguna librería, comencé con el diseño de la tabla de forma estática, añadiendo elementos que aún no disponían de funcionalidad alguna pero que serían útiles para las tablas.

- **Implementación del algoritmo de búsqueda en las tablas.** Tras la elaboración de la tabla estática y ya establecido un contrato JSON, me dispuse a implementar el algoritmo de búsqueda que modificaría el aspecto de la tabla.

- **Elaboración dinámica de la tabla básica mediante *jQuery* y posterior adaptación al script *table.js*.** Tras estar terminada la tabla al completo de forma estática comencé con la elaboración de la misma mediante *jQuery*, esta vez usando el contrato JSON.

- **Desarrollo de código encargado de enviar las peticiones mocks en el lado del cliente (servicios REST de AngularJS e inserción del comando correspondiente).** De forma a mi compañera, yo me dispuse a estudiar el funcionamiento de los servicios en AngularJS y el posterior desarrollo para realizar las peticiones de *mocks* al servidor.

- **Estudio y modificación del script *tree.js* para permitir el anidamiento.** En esta tarea debía adaptar el antiguo mecanismo de anidamiento de árboles a uno mejorado, haciendo uso de la librería elegida para permitir más tipos de representación mediante el uso de ventanas modales.

- **Elaboración de pruebas y depuración del anidamiento de los scripts *table.js* y *tree.js*.** Para ello, me elabore algunos ejemplos para comprobar el correcto funcionamiento del anidamiento, y si era necesaria, corregir el código.

- **Análisis de distintas figuras y luces para la creación las escenas tridimensionales.** En base a posibles usos de las escenas, pensé algunas figuras y luces que pudieran servir para tales propósitos apoyándome también en mi compañera.

- **Elaboración del contrato JSON de las escenas tridimensionales.** Mientras mi compañera realizaba el código de las escenas, me dispuse a elaborar el contrato *JSON* en base a las figuras y luces escogidas.

- **Inserción en las escenas del detector de tecnologías.** Dado que las escenas necesitan de unas tecnologías que deben estar disponibles en el navegador, me dispuse a elaborar el código necesario para realizar estas comprobaciones.

- **Inserción del parseo CSS en los elementos de las escenas.** Una vez elaboré las funciones necesarias, hice uso de ellas en las escenas para cambiar el aspecto visual de la misma.

- **Pruebas y depuración de *scene.js.***

- **Estudio, búsqueda y pruebas de alguna librería para crear menús contextuales.** Dado que pensamos en utilizar menús contextuales para la interacción y manipulación de las estructuras, me dediqué a investigar alguna librería útil para este objetivo.

- **Elaboración de las funciones necesarias en el fichero vz-context-menu.js.** En base a la estructura *JSON* definida, elaboré las funciones que construiría los menús contextuales y de las que haría uso los distintos *scripts* de las representaciones.

- **Modificación del script scene.js para el uso de menú contextuales.** Para ello, debía implementar un algoritmo para detectar los elementos seleccionados dentro de las escenas y mostrar el menú contextual asociado al mismo.

- **Elaboración de una de las funcionalidades de los *backreferences*.** En concreto, yo me dediqué a la encargada del procesamiento y limpieza de las expresiones utilizadas en los backreferences de interactividad, mediante la realización del código necesario en los distintos *scripts* de la aplicación como *vs-context-menu.js, wterm.js o backreferences.js*.

- **Elaboración de algunos ejemplos de representaciones gráficas.**

- **Escritura de algunos apartados de la memoria.** Entre ellos están el resumen, la introducción, la metodología y algunas secciones de contribuciones y extensibilidad.

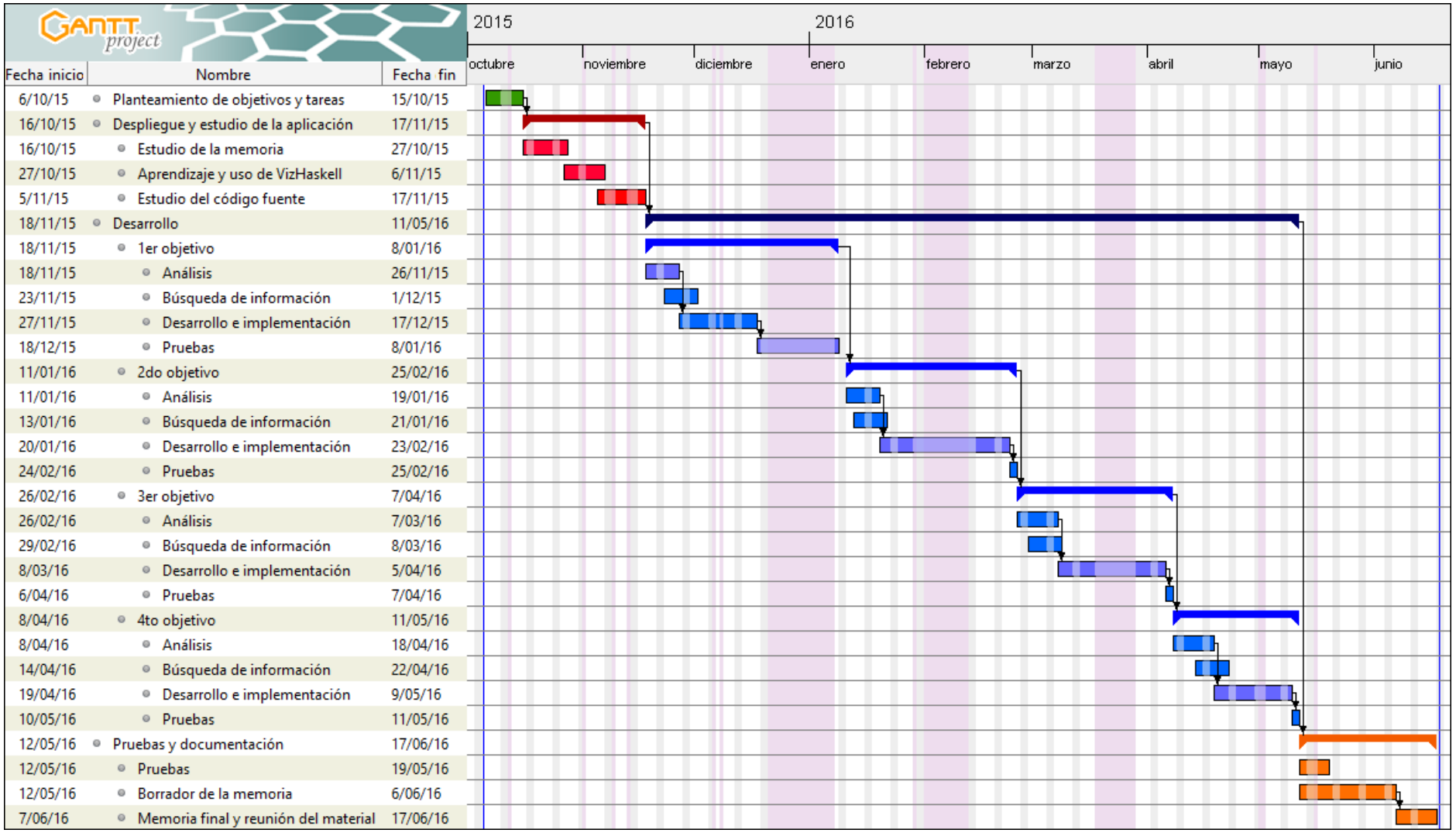
- **Inserción de las referencias y la bibliografía.**

- **Reunión del material digital y su grabado en DVD.**

Anexo I

Planificación de tareas

Nombre de la tarea	Fecha de inicio	Fecha de fin
Planteamiento de objetivos y tareas	6/10/15	15/10/15
Despliegue y estudio de la aplicación	16/10/15	17/11/15
Estudio de la memoria	16/10/15	27/10/15
Aprendizaje y uso de VizHaskell	27/10/15	6/11/15
Estudio del código fuente	5/11/15	17/11/15
Desarrollo	18/11/15	11/05/16
1er objetivo	18/11/15	8/01/16
Análisis	18/11/15	26/11/15
Búsqueda de información	23/11/15	1/12/15
Desarrollo e implementación	27/11/15	17/12/15
Pruebas	18/12/15	8/01/16
2do objetivo	11/01/16	25/02/16
Análisis	11/01/16	19/01/16
Búsqueda de información	13/01/16	21/01/16
Desarrollo e implementación	20/01/16	23/02/16
Pruebas	24/02/16	25/02/16
3er objetivo	26/02/16	7/04/16
Análisis	26/02/16	7/03/16
Búsqueda de información	29/02/16	8/03/16
Desarrollo e implementación	8/03/16	5/04/16
Pruebas	6/04/16	7/04/16
4to objetivo	8/04/16	11/05/16
Análisis	8/04/16	18/04/16
Búsqueda de información	14/04/16	22/04/16
Desarrollo e implementación	19/04/16	9/05/16
Pruebas	10/05/16	11/05/16
Pruebas y documentación	12/05/16	17/06/16
Pruebas	12/05/16	19/05/16
Borrador de la memoria	12/05/16	6/06/16
Memoria final y reunión del material	7/06/16	17/06/16

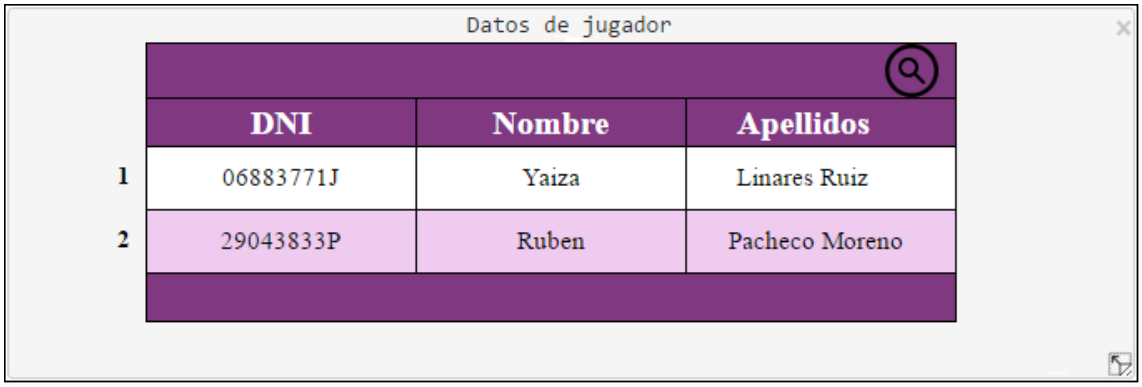


Fecha inicio	Nombre	Fecha fin
6/10/15	Planteamiento de objetivos y tareas	15/10/15
16/10/15	Despliegue y estudio de la aplicación	17/11/15
16/10/15	Estudio de la memoria	27/10/15
27/10/15	Aprendizaje y uso de VizHaskell	6/11/15
5/11/15	Estudio del código fuente	17/11/15
18/11/15	Desarrollo	11/05/16
18/11/15	1er objetivo	8/01/16
18/11/15	Análisis	26/11/15
23/11/15	Búsqueda de información	1/12/15
27/11/15	Desarrollo e implementación	17/12/15
18/12/15	Pruebas	8/01/16
11/01/16	2do objetivo	25/02/16
11/01/16	Análisis	19/01/16
13/01/16	Búsqueda de información	21/01/16
20/01/16	Desarrollo e implementación	23/02/16
24/02/16	Pruebas	25/02/16
26/02/16	3er objetivo	7/04/16
26/02/16	Análisis	7/03/16
29/02/16	Búsqueda de información	8/03/16
8/03/16	Desarrollo e implementación	5/04/16
6/04/16	Pruebas	7/04/16
8/04/16	4to objetivo	11/05/16
8/04/16	Análisis	18/04/16
14/04/16	Búsqueda de información	22/04/16
19/04/16	Desarrollo e implementación	9/05/16
10/05/16	Pruebas	11/05/16
12/05/16	Pruebas y documentación	17/06/16
12/05/16	Pruebas	19/05/16
12/05/16	Borrador de la memoria	6/06/16
7/06/16	Memoria final y reunión del material	17/06/16

Anexo II

Tabla simple 3x2

IMAGEN



ESTRUCTURA JSON

```
1. {
2.       "repType":"table",
3.       "label":"Datos de jugador",
4.       "columns":[
5.              {
6.                     "name":"DNI"
7.              },{
8.                     "name":"Nombre"
9.              },{
10.                    "name":"Apellidos"
11.             }
12.      ],
13.      "rows":[
14.             {
15.                    "key":{
16.                           "content":{
17.                                  "repType":"text",
18.                                  "text":"06883771J"
```

```
                        }
                },
                "values":[
                        {
                                "content":{
                                        "repType":"text",
                                        "text":"Yaiza"
                                }
                        },{
                                "content":{
                                        "repType":"text",
                                        "text":"Linares Ruiz"
                                }
                        }
                ]
        },{
                "key":{
                        "content":{
                                "repType":"text",
                                "text":"29043833P"
                        }
                },
                "values":[
                        {
                                "content":{
                                        "repType":"text",
                                        "text":"Ruben"
                                }
                        },{
                                "content":{
                                        "repType":"text",
                                        "text":"Pacheco Moreno"
                                }
                        }
                ]
        }
    ]
}
```

Anexo III

Escena simple

IMAGEN

ESTRUCTURA JSON

```
1.      "repType":"scene",
2.      "className":"purple",
3.      "lights":[
4.              {
5.                      "id":"light1",
6.                      "type":"pointlight",
7.                      "className":"blue",
8.                      "position":{
9.                              "y":400
10.                     }
11.             }
```

```
        ],
        "shapes":[
                {
                        "id":"text1",
                        "type":"text3D",
                        "className":"text",
                        "measures":{
                                "text":"VizHaskell"
                        },
                        "position":{
                                "y":200, "z":100
                        }
                },{
                        "id":"sphere1",
                        "type":"sphere",
                        "className":"sphere",
                        "measures":{
                                "radius":80
                        },
                        "position":{
                                "x":-300, "y":60, "z":10
                        },
                        "rotation":{
                                "x":0, "y":50, "z":0
                        }
                },{
                        "id":"prism1",
                        "type":"prism",
                        "className":"prism",
                        "measures":{
                                "base":{
                                        "type":"regPolygon",
                                        "measures":{
                                                "numSides":5,
                                                "sideLength":100
                                        }
                                },
                                "height":200
                        },
                        "position":{
                                "x":300, "y":20
                        },
                        "rotation":{
                                "x":180, "y":180
                        }
                }
        ]
}
```

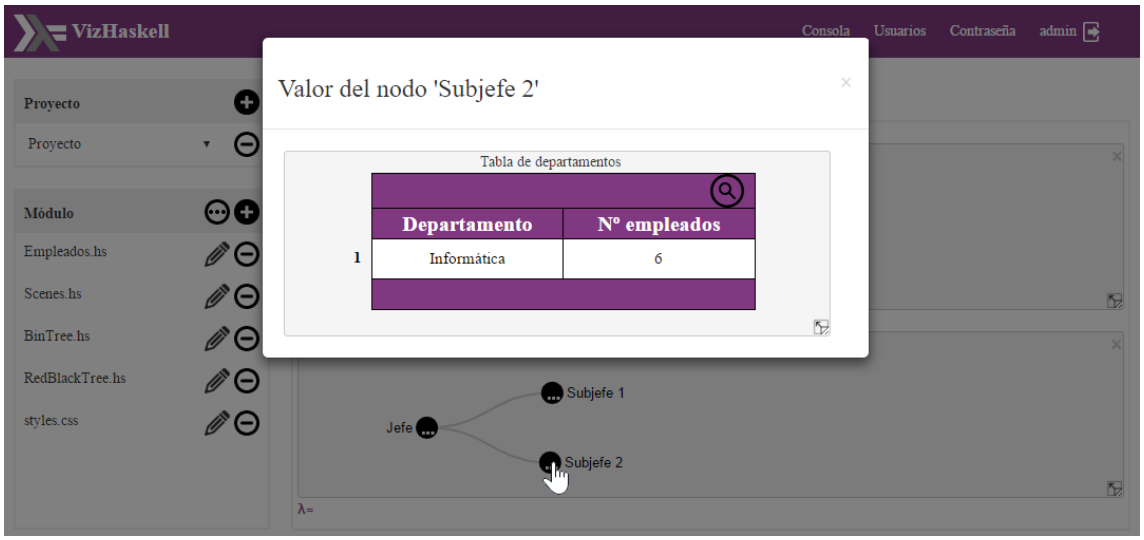
CSS

```
.purple {
  color: #35053C;
}
.blue {
  color: blue;
  intensity: 1;
  distance: 0;
}
.sphere {
  color: #ffc0cb;
  material: lambert;
}
.text {
  material: basic;
  color: #ef597b;
  font-size: 50;
}
```

Anexo IV

Anidamiento

IMAGEN



ESTRUCTURA JSON

```
1. {
2.         "repType":"tree",
3.         "label":"Jefe",
4.         "value":{
5.                 "content":{
6.                         "repType":"table",
7.                         "label":"Tabla de departamentos",
8.                         "columns":[
9.                                 {
10.                                        "name":"Departamento"
11.                                },{
12.                                        "name":"Nº empleados"
13.                                }
14.                        ],
15.                        "rows":[
16.                                {
17.                                        "key":{
```

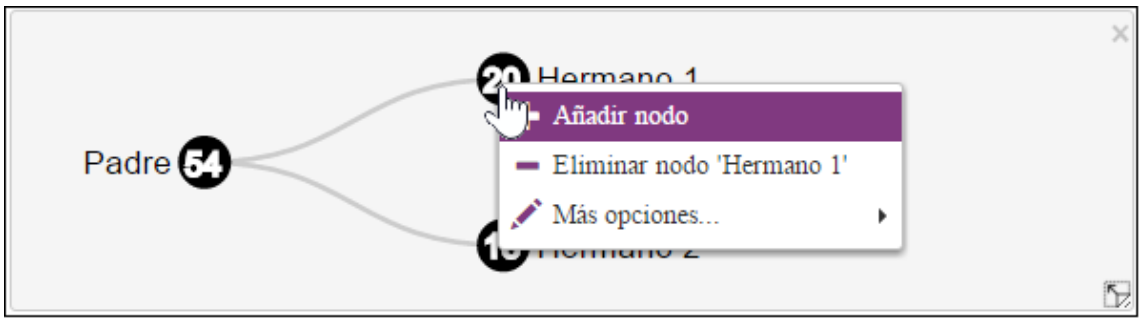
```
                                        "content":{
                                               "repType":"text",
                                               "text":"Ventas"
                                        }
                                    },
                                    "values":[
                                            {
                                                "content":{
                                                 "repType":"text",
                                                  "text":"25"
                                                 }
                                            }
                                     ]
                                }
                        ]
                }
        },
        "children":[
                {
                        "repType":"tree",
                        "label":"Subjefe 1",
                        "value":{
                                "content":{
                                        "repType":"table",
                                        "label":"Tabla de departamentos",
                                        "columns":[
                                                {
                                                    "name":"Departamento"
                                                },{
                                                    "name":"Nº empleados"
                                                }
                                        ],
                                        "rows":[
                                                {
                                                "key":{
                                                   "content":{
                                                       "repType":"text",
                                                       "text":"Marketing"
                                                    }
                                                },
                                                "values":[
                                                        {
                                                    "content":{
                                                       "repType":"text",
                                                       "text":"17"
                                                            }
                                                        }
                                                ]
                                            }
                                        ]
                                }
                        },
```

```
                    "children":[]
                },
                {
                    "repType":"tree",
                    "label":"Subjefe 2",
                   "value":{
                         "content":{
                                "repType":"table",
                                 "label":"Tabla de departamentos",
                                 "columns":[
                                          {
                                            "name":"Departamento"
                                          },{
                                            "name":"Nº empleados"
                                          }
                                   ],
                                  "rows":[
                                         {
                                           "key":{
                                             "content":{
                                               "repType":"text",
                                              "text":"Informática"
                                                 }
                                              },
                                       "values":[
                                           {
                                              "content":{
                                                "repType":"text",
                                                 "text":"6"
                                               }
                                           }
                                         ]
                                      }
                                 ]
                             }
                         },
                         "children":[]
                    }
            ]
    }
```

Anexo V

Interactividad

IMAGEN



ESTRUCTURA JSON

```
{
        "repType":"tree",
        "label":"Padre",
        "value":{
                "content":{
                        "repType":"text",
                        "text":"54"
                },
                "commands":[
                        {
                                "caption":"Añadir nodo",
                                "iconClass":"glyphicon glyphicon-plus",
                                "command":"insert ?MARC1 @@",
                                "parameters":[
                                        {
                                                "id":"MARC1",
                                                "caption":"Valor",
                                                "required":true
                                        }
                                ]
                        },{
```

```
                        "caption":"Eliminar Padre",
                        "command":"delete @@",
                        "iconClass":"glyphicon glyphicon-minus",
                         "parameters":[]
                    }
            ]
    },
    "children":[
            {
                    "repType":"tree",
                    "label":"Hermano 1",
                    "value":{
                            "content":{
                                    "repType":"text",
                                    "text":"20"
                            },
                        "commands":[
                               {
                            "caption":"Añadir nodo",
                            "iconClass":"glyphicon glyphicon-plus",
                            "command":"insert ?MARC1 @@",
                            "parameters":[
                                    {
                                        "id":"MARC1",
                                        "caption":"Valor",
                                        "required":true
                                      }
                                ]
                        },{
                           "command":"@@ delete Hermano 1",
                           "caption":"Eliminar 'Hermano 1'",
                           "iconClass":"glyphicon glyphicon-minus",
                            "parameters":[]
                            },{
                          "caption":"Más opciones...",
                         "iconClass":"glyphicon glyphicon-pencila",
                         "subcommands":[
                                    {
                                      "caption":"Redibujar árbol",
                                      "command":"@@",
                                      "parameters":[]
                                    }
                                       ],
                                            "parameters":[]
                                    }
                            ]
                    },
                    "children":[]
            },{
                    "repType":"tree",
                    "label":"Hermano 2",
                    "value":{
```

```
                                "content":{
                                        "repType":"text",
                                        "text":"18"
                                },
                        "commands":[
                                        {
                                "command":"@@ delete Hermano 2",
                                "caption":"Eliminar Hermano 2",
                                "iconClass":"glyphicon glyphicon-minus",
                                "parameters":[]
                                   },{
                                 "caption":"Añadir nodo",
                                 "iconClass":"glyphicon glyphicon-plus",
                                 "command":"insert ?MARC1 @@",
                                 "parameters":[
                                             {
                                              "id":"MARC1",
                                              "caption":"Valor",
                                              "required":true
                                               }
                                    ]
                                      }
                                ]
                        },
                        "children":[]
                }
        ]
}
```

Bibliografía

Baena, M. & Aragón, R. (2014) ***Visualización gráfica de tipos de datos Haskell***. http://eprints.sim.ucm.es/30199/.

Bhaumik, S. (2015) ***Bootstrap essentials: use the powerful features of bootstrap to create responsive and appealing web pages***. Birmingham, England: Packt Publishing. ISBN: 9781784395179 - 9781784396336 (e-book).

Cameron, D. (2015) ***HTML5, JavaScript, and jQuery 24-hour trainer***. Indianapolis, Indiana: Wrox. ISBN: 9781119001164 - 9781119001188 (e-book).

Cochran, D. (2012) ***Twitter Bootstrap Web Development How-To***. England: Packt Publishing, Limited, 2012. ISBN: 9781849518826.

Dirksen, J. (2015) ***Learning Three.js: The JavaScript 3D Library for WebGL (2nd ed.)***. Birmingham, England; Mumbai, India: Packt Publishing. ISBN: 9781784392215 - 9781784391027 (e-book).

Dirksen, J. (2015) ***Three.js cookbook: over 80 shortcuts, solutions, and recipes that allow you to create the most stunning visualizations and 3D scenes using the Three.js library***. Birmingham, England: Packt Publishing. ISBN: 9781783981182 - 9781783981199 (e-book).

Dirksen, J. (2014) ***Three.js essentials: create and animate beautiful 3D graphics with this fast-paced tutorial***. Birmingham, England: Packt Publishing. ISBN: 9781783980864 - 9781783980871 (e-book).

GHC Team (2007) ***GHC Users Guide Documentation***, Version 8.0.1. https://downloads.haskell.org/~ghc/latest/docs/users_guide.pdf

Knol, A. (2013) ***Dependency Injection with AngularJS***. Birmingham, England: Packt Publishing. ISBN: 9781782166566 - 9781782166573 (e-book).

Meeks, E. (2015) ***D3.js in action*** Shelter Island, NY: Manning. ISBN: 9781617292118

Prettyman, S. (2016) ***Learn PHP 7: Object-Oriented Modular Programming using HTML5, CSS3, JavaScript, XML, JSON, and MySQL.*** Berkeley, CA: Apress. ISBN: 9781484217306

Raasch, J.J. & others (2015) ***Javascript and jQuery for data analysis and visualization***. Indianapolis, Indiana: John Wiley and Sons. ISBN: 9781118847060 - 9781118847213 (e-book).

Rantala, M. (2015) ***Real-time collaborative coding - technical and group work challenges.***
http://dspace.cc.tut.fi/dpub/bitstream/handle/123456789/23112/rantala.pdf

Sriparasa, S.S. (2013) ***JavaScript and JSON essentials***. Birmingham, England: Packt Publishing. ISBN: 9781783286034 - 9781783286041 (e-book)

Williamson, K. (2015) ***Learning AngularJS***. ISBN: 9781491916742 (e-book)

Recursos

Brala, B. (2011)***_jQuery-contextMenu - JQuery contextMenu plugin & polyfill***. MIT license. http://swisnl.github.io/jQuery-contextMenu

Cabello, R. (2010) ***Three - JavaScript 3D library***. MIT license. http://threejs.org/.

Dante (2013) ***Bootstrap-dialog - Make use of Bootstrap Modal more monkey-friendly***. MIT license. https://github.com/nakupanda/bootstrap3-dialog.

Glazman, D. (2010) ***JSCSSP - a CSS parser in JavaScript***. MPL/GPL/LGPL triple license. https://github.com/therealglazou/jscssp.

W3C ***Documentation and specification of HTML5*** https://www.w3.org/TR/html5/

W3C ***Documentation and specification of CSS*** https://www.w3.org/TR/#tr_CSS

ECMA (2013) ***Standard ECMA-404 - The JSON Data Interchange Format 1st Edition*** http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

Miscelánea

https://jsonformatter.curiousconcept.com/

http://json.parser.online.fr/

http://www.jlabstudio.com/webgl/category/programacion/webgl/three-js/

https://msdn.microsoft.com/es-es/library/dn479430(v=vs.85).aspx

http://stackoverflow.com/questions/27193732/three-js-creating-a-right-triangular-prism

http://alexan0308.github.io/threejs/examples/#webgl_geometry_PrismGeometry

http://www.ibm.com/developerworks/library/wa-webgl2/

http://stackoverflow.com/questions/29809646/debug-threejs-raycaster-mouse-coordinates

http://stemkoski.github.io/Three.js/

http://www.genbetadev.com/javascript/introduccion-a-three-js-la-libreria-3d-numero-uno-para-html5

https://github.com/mrdoob/three.js/wiki/Drawing-lines

http://jsfiddle.net/theo/Vc7yr/

http://stackoverflow.com/questions/13231539/how-make-3d-text-in-three-js

https://gist.github.com/mrflix/8351020

http://www.lab4games.net/zz85/blog/2014/09/08/rendering-lines-and-bezier-curves-in-three-js-and-webgl/

http://www.johannes-raida.de/tutorials/three.js/tutorial02/tutorial02.htm

http://www.tutosytips.com/dia-13-dibujemos-formas-basicas-con-canvas/

http://qiku.es/pregunta/42181/c%C3%B3mo-crear-una-serie-de-l%C3%ADnea-threejs-3d-con-anchura-y-espesor-how-to-create-a-threejs-3d-line-series-with-width-and-thickness

http://qiku.es/pregunta/314820/extrusion-una-linea-en-threejs-extruding-a-line-in-threejs

http://stackoverflow.com/questions/12656138/how-to-render-2d-shape-of-points-in-three-js