Haskell.do, un entorno de desarrollo integrado enfocado hacia la ciencia de datos y el desarrollo interactivo para el lenguaje de programación Haskell

Nikita Tchayka Razumov

Junio de 2018

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Escuela de Ingeniería Informática

TFT04

SOLICITUD DE DEFENSA DE TRABAJO DE FIN DE TÍTULO

D/Dª Nikita Tchayka Razumov, autor del Trabajo de Fin de Título Haskell.do, un entorno de desarrollo integradoenfocado hacia la ciencia de datos y el desarrollo interactivo para el lenguaje de programación Haskell, correspondiente a la titulación Grado en Ingeniería Informática, en colaboración con la empresa/proyecto (indicar en su caso) The Agile Monkeys S.L.

S O L I C I T A

que se inicie el procedimiento de defensa del mismo, para lo que se adjunta la documentación requerida.

Asimismo, con respecto al registro de la propiedad intelectual/industrial del TFT, declara que:

[X] Se ha iniciado o hay intención de iniciarlo (defensa no pública).
[ ] No está previsto.

Y para que así conste firma la presente.

Las Palmas de Gran Canaria, a 16 de julio de 2018.

El estudiante

Fdo.: _____

A rellenar y firmar **obligatoriamente** por el/los tutor/es

En relación a la presente solicitud, se informa:

[;] Positivamente

[ ] Negativamente
(la justificación en caso de informe negativo deberá incluirse en el TFT05)

Fdo.: Firmado por SALGADO DE LA NUEZ AGUSTIN JAVIER - 54072806J el día 17/07/2018 con un certificado emitido por

DIRECTOR DE LA ESCUELA DE INGENIERÍA INFORMÁTICA

Agradecimientos

Agradezco a todas aquellas personas que han pasado por mi vida y han tenido algún contacto conmigo, sea bueno, o malo. Ustedes habéis esculpido la persona que soy hoy en día.

Gracias a Agustín, por ayudarme con este documento. Su ayuda ha sido muy importante, ya que es mi primera vez redactando un documento de este estilo.

Gracias a mi familia, por haberme permitido tener la educación que tengo.

Gracias a mis amigos y compañeros, que han hecho que la experiencia de la Universidad valiera mucho más la pena.

Gracias a Theam, The Agile Monkeys, o cualquiera de sus futuros nombres. Gracias a cada uno de sus miembros por hacer de la empresa la mejor empresa para trabajar. Sin ustedes no tendría la misma motivación con la que me levanto todos los días.

Y sobre todo gracias a Verónica. Amiga incondicional, pareja, esposa, psicóloga, compañera, artista y apoyo moral.

Mi mayor felicidad no es mi carrera, sino tener a una persona como tú a mi lado.

Esta página ha sido intencionalmente dejada en blanco.

Índice general

Capítulo 1

Introducción

Este proyecto se centra en la planificación, diseño y desarrollo de un entorno de desarrollo, centrado en el desarrollo interactivo. Se presenta en formato de aplicación web open source, alojada en la plataforma GitHub. Está principalmente orientado a desarrolladores, analistas y científicos de datos que utilicen el lenguaje de programación Haskell como tecnología principal en su trabajo diario.

El proyecto se ha presentado en diversas conferencias de Madrid, Cádiz y Zúrich, Suiza. Actualmente cuenta con 300 estrellas en GitHub.

El proyecto está implementado basado en estándares de diseño y arquitectura funcional, como son **transformadores de mónadas**, **functores aplicativos** y **programación reactiva**.

Se presenta como una aplicación distribuida, permitiendo ejecutar código en sistemas remotos como pueden ser centros de datos, aprovechando la capacidad de cómputo de estos. El usuario puede utilizar la aplicación desde cualquier dispositivo que tenga un navegador web y conexión a Internet.

Durante el proceso de desarrollo, se ha utilizado una metodología Scrum para gestionarlo. Esta es una metodología ágil. Estas ofrecen un enfoque para la toma de decisiones en los proyectos software, que se refiere a métodos de ingeniería del software basados en el desarrollo iterativo e incremental, donde los requisitos y soluciones evolucionan con el tiempo según las necesidades del proyecto. Así, el trabajo es realizado y distribuido por equipos o individuos auto-organizados, inmersos en un proceso compartido de tomas de decisiones a corto plazo.

A la hora de desarrollar el código en si, se ha seguido la metodología Type

Driven Development. Que se basa en especificar todas las estructuras de datos posibles para las diferentes características del software. De esta manera se consiguen que todos los estados del software estén representados, y que los erróneos no sean posibles. Gracias a esto, los errores de tiempo de ejecución se desplazan al tiempo de compilación y es el compilador, quien se encarga de avisar al desarrollador de que el código puede incurrir en errores.

Cada iteración sobre el producto incluye:

1. Planificación
2. Análisis de los requisitos
3. Diseño
4. Codificación
5. Documentación

El objetivo de cada iteración no ha sido el generar todas las características necesarias para lanzar el producto, sino, incrementar el valor por medio de software funcional.

A pesar de que se ha seguido una metodología Scrum, la comunidad Open Source ha ayudado a detectar incidencias y a priorizar correctamente las características.

Finalmente, la memoria se estructura de la siguiente manera:

Estado del arte: Análisis de aplicaciones que pertenezcan al mismo ámbito que Haskell.do y las razones de la creación de este proyecto.

Objetivos: Objetivos académicos y personales de este proyecto.

Competencias: Competencias obtenidas tras la realización de este proyecto.

Recursos: Tecnologías utilizadas.

Metodologías: Repaso por las metodologías aplicadas en el desarrollo y planificación del proyecto.

Planificación: Gestión de los tiempos de entrega y organización temporal en el desarrollo.

Sprints: Detalle del proceso realizado durante cada uno de los sprints.

Colofón: En este capitulo se discutirán las conclusiones, aportaciones del proyecto a mi aprendizaje y la conclusión del producto. Además se hablará de las posibles mejores mejoras futuras.

Detalles técnicos: Funcionamiento interno de Transient y de las tecnologías mas importantes.

Bibliografía: Recursos utilizados para el desarrollo de este proyecto.

Capítulo 2

Estado del arte

En este capítulo se realizará un breve estudio sobre aplicaciones en el campo de las herramientas de desarrollo para ciencias de datos. Tanto para otros lenguajes como para Haskell.

2.1 Python

Actualmente, Python se posiciona como el lenguaje mas utilizado en ciencia de datos:

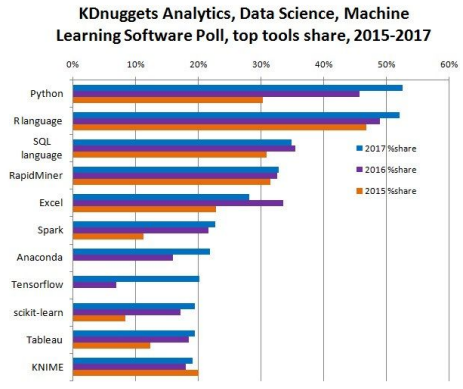

Figura 2.1: Popularidad de Python frente a otras herramientas de ciencia de datos[1]

Es por ello por lo que sus herramientas son una gran inspiración:

1. **Jupyter (antiguamente IPython notebook)**

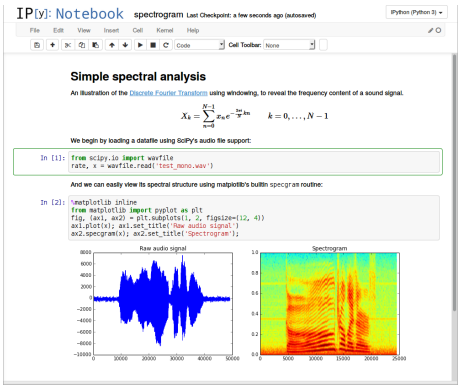

Figura 2.2: Captura de un flujo de trabajo en Jupyter Notebook

Es un entorno de desarrollo basado en una interfaz web. La ventaja que ofrece Jupyter frente a otras soluciones similares es que es una solución distribuida, permitiendo tener el núcleo de ejecución en un sistema de alta capacidad, mientras que el usuario maneja la interfaz gráfica desde su sistema, que no tiene porque ser de alto rendimiento.

Jupyter está basado en el concepto de *celda*, lo cual es su modelo de ejecución principal, como se muestra en la Figura 2.2. Una celda puede ser de dos tipos, de texto o de código.

Una celda de texto es aquella donde podemos escribir ecuaciones LaTeX y explicaciones en un formato de texto plano, permitiéndonos documentar el código con comentarios más enriquecidos.

Una celda de código es un bloque de código a interpretar. Jupyter es lo suficientemente inteligente para saber si esa celda hace declaraciones o devuelve un resultado. En el caso de devolver un resultado que es una gráfica. La dibuja en el propio documento.

2. **Spyder**

Spyder es una aplicación de escritorio basada en *IPython* (núcleo de Jupyter). El usuario tiene un archivo de código y una consola para probar cosas. A su vez, una tabla de variables que muestra los valores de ellas. Por último, existe un explorador de documentación a disposición del usuario.

La mayor diferencia con Jupyter, es que Spyder no ofrece una experiencia integrada, donde todo está en el mismo documento, sino que es como un campo de pruebas anárquico donde es el propio usuario quien

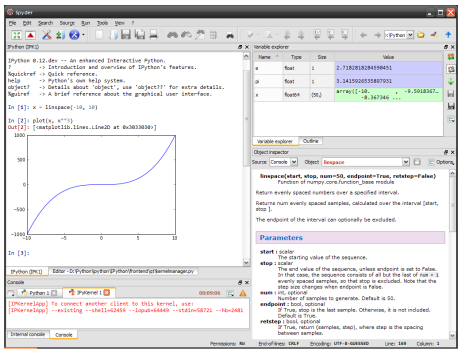

Figura 2.3: Captura de un flujo de trabajo en Spyder

decide que guardar y que no.

2.2 R

R, como se muestra en la Figura 2.1, es el segundo lenguaje de programación más utilizado para ciencia de datos. Se diseñó inicialmente como un lenguaje de programación orientado a análisis de datos, por lo tanto posee de muy buenas librerías y paquetes.

1. **R Studio**

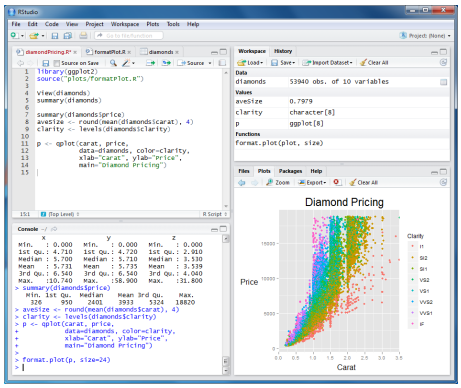

Figura 2.4: Captura de un flujo de trabajo en R Studio

Es un IDE, muy parecido a Spyder, pero sobre la plataforma R. Posee todas las características de este, pero con el detalle de que la plataforma a utilizar es R y no Python.

2.3 Markdown

A pesar de que no es ni siquiera un lenguaje de programación, Markdown, siendo un lenguaje de marcado extremadamente simple, ha conseguido una popularidad increíble a la hora de escribir documentación.

Su sintaxis sencilla permite escribir documentación sin preocuparnos de las opciones de formato. Por lo tanto lo hace una herramienta muy potente para redactar informes de cualquier tipo, incluyendo de ciencia de datos.

Existen varias herramientas de Markdown, pero solo nos centraremos en Markdown Pad.

1. **Markdown Pad**

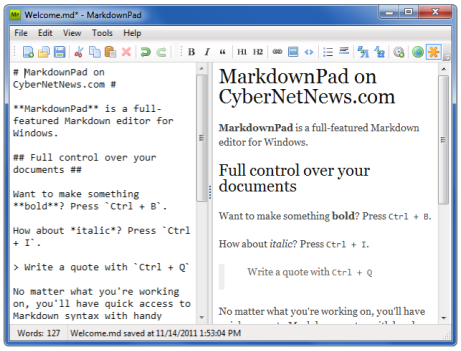

Figura 2.5: Captura de un flujo de trabajo en Markdown Pad

Como se ve en la Figura 2.5, este entorno permite escribir Markdown como texto plano a la izquierda y obtener el resultado en vivo a la derecha.

2.4 Haskell

Haskell es un lenguaje relativamente antiguo, puesto que Python y Haskell aparecieron el mismo año. A pesar de su edad, Haskell no ha sido tan popular como Python, y menos para ciencia de datos, por lo tanto no se han generado la misma cantidad de herramientas que en Python. Aparte de los típicos editores de texto orientados a desarrollo de propósito general, como son Emacs, Vim o Atom, existen unas soluciones para ciencia de datos no muy maduras.

1. **IHaskell notebook**

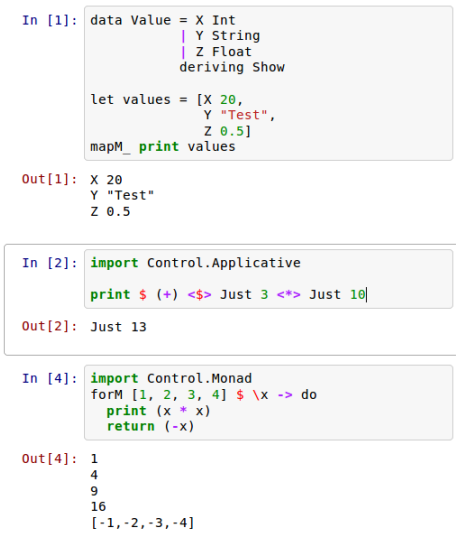

Figura 2.6: Captura de un flujo de trabajo en IHaskell notebook

IHaskell es un entorno de desarrollo de ciencia de datos que extiende Jupyter para añadirle soporte para Haskell. A pesar de que funciona relativamente bien, es muy frágil y la aplicación se congela cuando se ejecutan algoritmos no triviales. También, la instalación es un proceso muy tedioso, el cual se ha intentado resolver utilizando Docker, pero de esta manera se ha perdido la posibilidad de añadir paquetes a nuestro proyecto.

2. **HyperHaskell**

HyperHaskell es un editor open source mantenido por una única persona. El proyecto no es muy estable, puesto que se realizan contribuciones rara vez. Otra importante desventaja es que no permite sentencias imperativas como IHaskell, sino que todo el código tiene que ser una expresión funcional para evaluarse. Esto tiene una importante desventaja, y es que no se pueden guardar valores en variables, ni tampoco crear tipos. Convirtiéndose en una consola ligeramente más gráfica, ya que tiene resaltado de sintaxis y permite mostrar gráficos.

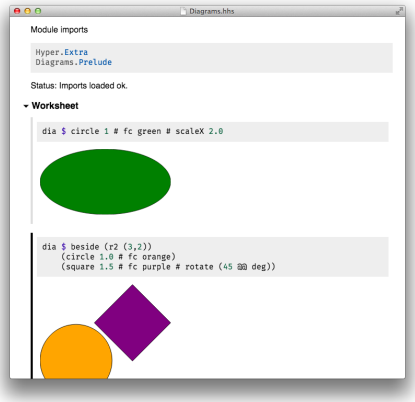

Figura 2.7: Captura de un flujo de trabajo en Hyper Haskell

2.5 Resumen

El estado del arte de los entornos de desarrollo interactivos para ciencia de datos es Jupyter principalmente, es con diferencia el editor más utilizado tanto profesionalmente, como académicamente.

Sin embargo, en Haskell, el paisaje es completamente yermo y no existe ninguna solución parecida a esta.

¿Por qué Haskell?

Haskell es un lenguaje funcional puro de propósito general, que muchísimo en la programación del día a día. El compilador de Haskell más utilizado, GHC, es increíblemente potente, ya que analiza todas las posibilidades de error del código en tiempo de compilación.

Aparte de esto, es un lenguaje muy declarativo, evitándole al programador el hecho de tener que especificar lo que tiene que hacer el sistema constantemente. ***Coge el valor 42, guárdalo en la variable X, ...*** . Es un lenguaje que permite expresar las operaciones de tratamiento de datos de una manera muy concisa y sencilla.

Además es un lenguaje compilado a código máquina, cuyo rendimiento generalmente es muy similar al lenguaje C++, mientras que Python es un lenguaje mucho más lento en comparación a Haskell, y es por ello por lo que Python siempre depende de librerías nativas, escritas en C o FORTRAN, para poder aprovechar la velocidad del computador.

Haskell no tiene este problema, ya que el algoritmo que escribamos va a

ser generalmente optimizado por el compilador automáticamente y posteriormente convertido a código máquina. Por ejemplo, el algoritmo de C y la función recursiva escrita en Haskell a continuación, generan el mismo código de ensamblador X86:

```c
int i;
int result = 0;
for (i = 0; i < 10; i++) {
  result = result + 1;
}
```

Listing 1: Algoritmo en C

```haskell
myFunction 1 = 1
myFunction i =
  1 + myFunction (i - 1)

result = myFunction 10
```

Listing 2: Función recursiva en Haskell

Es por ello, por lo que Haskell es un lenguaje ideal para la ciencia de datos y es importante potenciar su capacidad para este campo de las ciencias de la computación.

Capítulo 3

Objetivos

3.1 Objetivos académicos

Tal como se especifica en el Reglamento General para la realización y evaluación de trabajos de fin de título, la realización de un Trabajo Fin de Grado tiene por objetivo elaborar un trabajo en el que el estudiante universitario desarrolle las competencias y los conocimientos adquiridos, teóricos y prácticos como culminación de sus estudios y como preparación para el desempeño futuro de actividades profesionales en el ámbito correspondiente a la titulación obtenida.

3.2 Objetivos generales del proyecto

Teniendo en cuenta las posibles dificultades que puede entrañar este proyecto, se plantea de acuerdo a los siguientes objetivos:

1. Aprendizaje, familiarización y uso de patrones de arquitectura funcional y programación distribuida.
2. Definir los requisitos del sistema, y sus posibles casos de uso.
3. Diseñar y desarrollar una interfaz gráfica de acceso y manejo de la información, que contemplará parámetros de usabilidad
4. Definir un flujo de datos basado en paso de mensajes y el modelo de actores.

5. Elaboración de la documentación mínima necesaria en cada una de las fases para su posterior uso incluyendo las fases de puesta en producción.
6. Uso de Scrum para el desarrollo y coordinación del proyecto.

Desde el punto de vista del trabajo a realizar, se persiguen las siguientes funcionalidades:

1. Interfaz gráfica acorde a la tarea de análisis de datos.
2. Carga y guardado de proyectos.
3. Evaluación de código.
4. Gestión de dependencias.

Capítulo 4

Competencias

4.1 CII01

Capacidad para diseñar, desarrollar, seleccionar y evaluar aplicaciones y sistemas informáticos, asegurando su fiabilidad, seguridad y calidad, conforme a principios éticos y a la legislación y normativa vigente.

Esta competencia queda cubierta con los capítulos de Introducción, Planificación, Recursos y en los apartados de cada Sprint: Sprint 1, Sprint 2 y Sprint 3. En ellos se especifican, por un lado, el motivo de la elección de esta aplicación para llevarla a cabo como TFG y, por otro lado, el análisis de tecnologías y recursos a utilizar para llevarlo a cabo satisfactoriamente.

4.2 CII02

Capacidad para planificar, concebir, desplegar y dirigir proyectos, servicios y sistemas informáticos en todos los ámbitos, liderando su puesta en marcha y su mejora continua y valorando su impacto económico y social.

De todas las tareas a realizar por un Ingeniero del Software, la planificación es, posiblemente, la más importante, puesto que una mala o inexistente planificación puede repercutir inevitablemente a una mala realización del proyecto, además de afectar los plazos de entrega, ocasionando grandes perjuicios. Por ello, esta competencia queda cubierta en los capítulos de Introducción, Planificación y en los apartados de cada Sprint, donde se detalla desde la Pila de Producto, que varía según las necesidades descubiertas a lo largo del

desarrollo de la aplicación, como en la especificación de cada Sprint, dando al final de cada uno un producto apto para entregar al supuesto cliente.

4.3 CII03

Capacidad para comprender la importancia de la negociación, los hábitos de trabajo efectivos, el liderazgo y las habilidades de comunicación en todos los entornos de desarrollo de software.

Antes de comenzar el desarrollo del proyecto, ya se había avistado una situación adversa en el panorama de la ciencia de datos en Haskell. Existían muchas librerías si, pero no existía un estándar como el que ofrece Python, ya que al contrario que Haskell, es un lenguaje muy acotado y limitado. Haskell, siendo un lenguaje muy expresivo, permite una variedad de APIs e integraciones inimaginable, complicando la conexión entre distintas librerías.

Por si esto no fuera poco, la gran mayoría de estas librerías estaba sin documentar, lo que hacía su uso aun más complicado.

Es por esto por lo que decidí fundar la organización de dataHaskell en septiembre de 2016, un equipo de voluntarios para mejorar este entorno con documentación y código abierto.

Esta competencia queda cubierta gracias al *feedback* recibido para mejorar el proyecto, y la capacidad de negociar *trade-offs* con la comunidad con el objetivo de agilizar el proceso.

4.4 CII06

Conocimiento y aplicación de los procedimientos algorítmicos básicos de las tecnologías informáticas para diseñar soluciones a problemas, analizando la idoneidad y complejidad de los algoritmos propuestos.

Como se verá más adelante, los algoritmos de evaluación de código se han ido mejorando paulatinamente, optimizando y mejorando en cada momento. Además, se ha diseñado la interfaz de usuario con una arquitectura reactiva, donde se ha tenido que plantear un propio motor de gestión de eventos.

4.5 CII07

Conocimiento, diseño y utilización de forma eficiente los tipos y estructuras de datos más adecuados a la resolución de un problema.

Haskell es un lenguaje fuertemente tipado, con un sistema de tipos que es lo suficientemente expresivo como para incluso programar en él. Esto se debe a que se basa en el *Isomorfismo de Curry-Howard*. [1] Durante el transcurso del proyecto se utilizan constantemente estructuras de datos basadas en los *tipos de datos algebraicos* [2]

4.6 CII08

Capacidad para analizar, diseñar, construir y mantener aplicaciones de forma robusta, segura y eficiente, eligiendo el paradigma y los lenguajes de programación más adecuados.

Esta competencia queda cubierta con el análisis hecho en el apartado de introducción. Se ha resaltado las razones de por qué Haskell es el lenguaje de programación idóneo para este problema y los patrones a utilizar en el transcurso del desarrollo para asegurar la estabilidad y robustez del proyecto.

4.7 CII011

Conocimiento y aplicación de las características, funcionalidades y estructura de los Sistemas Distribuidos, las Redes de Computadores e Internet y diseñar e implementar aplicaciones basadas en ellas.

La arquitectura del proyecto se ha planteado como un sistema distribuido,

[1] El isomorfismo de Curry-Howard, también conocido como la correspondencia de proposiciones lógicas como tipos, es una generalización donde se une la lógica formal y los sistemas de tipos para modelos de computación, permitiendo representar cualquier proposición lógica y verificación formal con un tipo. `https://en.wikipedia.org/wiki/Curry%E2%80%93Howard_correspondence`

[2] Un tipo de datos algebraico es un tipo de tipo compuesto, formado por dos estructuras comunes, tipos producto, representados como A × B y tipos suma, representados como A+B. Permiten codificar de una manera muy expresiva los modelos de datos de un determinado dominio.

donde tanto el servicio de ejecución, como el de interfaz de usuario pueden ejecutarse en dos sistemas completamente distintos con la capacidad de incluso distribuir el servicio de ejecución en diversos nodos para mejorar el rendimiento de las aplicaciones del usuario.

En los siguientes apartados se explicará la arquitectura y como se ha llevado a cabo utilizando un modelo de programación implícita.

4.8 CII013

Conocimiento y aplicación de las herramientas necesarias para el almacenamiento, procesamiento y acceso a los Sistemas de información, incluidos los basados en web.

En los siguientes apartados se explicará como se han utilizado los estándares web para generar la interfaz de usuario e interoperar con los distintos servicios del proyecto.

4.9 CII014

Conocimiento y aplicación de los principios fundamentales y técnicas básicas de la programación paralela, concurrente, distribuida y de tiempo real.

En el modelo de programación implícita utilizado, se gestiona la concurrencia y distribución de la aplicación de una manera especial, no común en otros paradigmas y/o lenguajes. En secciones posteriores se explicará en detalle.

4.10 CII016

Conocimiento y aplicación de los principios, metodologías y ciclos de vida de la ingeniería de software.

En el transcurso del proyecto se ha aplicado Scrum, una metodología de desarrollo ágil, entendiéndola en profundidad para poder organizar el proyecto. Por otro lado, la elección de Scrum no fue al azar, sino que se eligió como la mejor metodología para este proyecto frente a otras como Waterfall.

4.11 IS01

Capacidad para desarrollar, mantener y evaluar servicios y sistemas software que satisfagan todos los requisitos del usuario y se comporten de forma fiable y eficiente, sean asequibles de desarrollar y mantener y cumplan normas de calidad, aplicando las teorías, principios, métodos y prácticas de la ingeniería del software.

Mayoritariamente, el proyecto está verificado en base a la correspondencia de proposiciones como tipos y es el compilador quien se encarga de mantener un nivel aceptable de fiabilidad. Se han aplicado los patrones y principios funcionales en todo momento.

4.12 IS03

Capacidad de dar solución a problemas de integración en función de las estrategias, estándares y tecnologías disponibles

Se ha realizado un estudio de las librerías y tecnologías disponibles en el primer Sprint. Como se verá en esa sección, el estudio ha sido bastante detallado y esta competencia queda cubierta.

4.13 IS04

Capacidad de identificar y analizar problemas y diseñar, desarrollar, implementar, verificar y documentar soluciones software sobre la base de un conocimiento adecuado de las teorías, modelos y técnicas actuales.

EL código del proyecto se basa completamente en teorías, modelos y técnicas funcionales, a menudo derivadas de ramas de las matemáticas, como la lógica o la teoría de categorías. El código pretende ser auto-documentado, y además, lo suficientemente expresivo como para ser verificado formalmente estáticamente.

4.14 CP02

Capacidad para conocer los fundamentos teóricos de los lenguajes de programación y las técnicas de procesamiento léxico, sintáctico y semántico asociadas, y saber aplicarlas para la creación, diseño y procesamiento de lenguajes.

En los primeros Sprints se utilizaron técnicas de procesamiento de lenguajes basadas en parsers LL. En sus respectivas secciones se detallará.

4.15 CP03

Capacidad para evaluar la complejidad computacional de un problema, conocer estrategias algorítmicas que puedan conducir a su resolución y recomendar, desarrollar e implementar aquella que garantice el mejor rendimiento de acuerdo con los requisitos establecidos.

A menudo, en los lenguajes funcionales, es fácil caer en algoritmos subóptimos debido a como están implementadas las funciones estándar. En secciones posteriores veremos como se utilizan funciones que utilizan técnicas de *fusión* [3]

4.16 CP06

Capacidad para desarrollar y evaluar sistemas interactivos y de presentación de información compleja y su aplicación a la resolución de problemas de diseño de interacción persona-computadora.

La interfaz de usuario ha sido refinada durante varias iteraciones para ofrecer la mejor experiencia posible, centrando el tiro en la simplicidad y el minimalismo. Posteriormente se hará un análisis detallado del progreso.

4.17 SI03

Capacidad para participar activamente en la especificación, diseño, implementación y mantenimiento de los sistemas de información y comunicación.

[3] Una fusión de funciones, o una función de fusión es cuando estáticamente se optimiza

Como se ha mencionado anteriormente, la comunidad open source **dataHaskell** ha participado activamente en el diseño del software y se ha conseguido mantener un software lo suficientemente estable como para que hoy en día siga siendo una solución usable.

4.18 SI06

Capacidad para comprender y aplicar los principios y las técnicas de gestión de la calidad y de la innovación tecnológica en las organizaciones.

Como fundador de dataHaskell, durante el desarrollo del proyecto he tenido que gestionar reuniones y documentación para poder apoyar a la comunidad en su crecimiento. En apenas dos años, se ha pasado de ser 50 miembros, a ser casi 340. Esto ha sido posible gracias a la priorización de tareas y gestión de esfuerzo llevado a cabo regularmente.

una composición de funciones puras, reduciendo la complejidad de la función compuesta al orden de la función componente más compleja. *Stream Fusion - Coutts et al. [2]*

Capítulo 5

Recursos

5.1 Software utilizado

5.1.1 NixOS como sistema operativo

NixOS es una distribución de GNU/Linux basado en el gestor de paquetes Nix, que permite crear entornos de trabajo reproducibles. Está basado en la filosofía pura de programación funcional.

5.1.2 Emacs

Emacs es un editor de texto creado por GNU con código abierto y filosofía libre basado en la licencia GPL. Permite crear un entorno de desarrollo integrado (IDE) perfecto para Haskell.

5.1.3 Firefox

Firefox es un navegador web libre desarrollado por la Mozilla Foundation.

5.1.4 Stack

Stack es una herramienta de construcción de proyectos para Haskell, basado en la filosofía de estabilidad. Posee su propio repositorio de paquete don-

de se comprueba que librerías de Haskell producen incompatibilidades y las eliminan del repositorio.

5.1.5 Cabal

Cabal es el gestor de paquetes de Haskell, permite la descarga automática de librerías y aplicaciones Haskell.

5.2 Lenguajes de programación

5.2.1 Haskell

Haskell es un lenguaje de programación compilado. Se define como:

- Funcional puro: Toda función de Haskell (a excepción de aquellas de tipo `IO`) devuelven la misma salida para la misma entrada.
- Permite programación imperativa: Gracias al concepto de Mónada, Haskell permite codificar la programación imperativa en el paradigma de la programación funcional.
- Efectos secundarios controlados: a menos que se especifique explícitamente en el tipo, es imposible que una función pura ejecute una sentencia de `print`, por ejemplo.
- Estrictamente tipado estáticamente: Haskell posee uno de los sistemas de tipos más estrictos y flexibles, solo siendo superado por lenguajes como Coq, Agda e Idris. Evita el 90 % de los errores de ejecución en tiempo de compilación.
- Con soporte para tipos de orden superior: Permite representar tipos del tipo `* ->*`, donde se especifica que un tipo genérico ha de ser estar parametrizado por tipo genérico.
- Con soporte para polimorfismo ad-hoc: Permite extender tipos existentes en base a interfaces, fuera de su declaración. Esto no es posible en Java, ya que la implementación de la interfaz se hace en la implementación del tipo.

5.2.2 HTML

HTML es un lenguaje de marcado para la elaboración de páginas web. Es un estándar que sirve de referencia del software que conecta con la elaboración de páginas web en sus diferentes versiones, define una estructura básica y un código (denominado código HTML) para la definición de contenido de una página web, como texto, imágenes, vídeos y juegos, entre otros.

5.3 CSS

CSS es un lenguaje usado para definir el estilo de un documento escrito en HTML. Esta herramienta permite separar la estructuración de un documento de su presentación. Como nota interesante, HTML y CSS son *turing-complete*, permitiendo hacer todo tipo de algoritmos de estilo y/o estructuración con estas dos herramientas.

5.4 Librerías y Frameworks

5.4.1 Transient

Transient es un framework de Haskell, escrito por Alberto Gómez Corona. Permite al desarrollador escribir de una manera sencilla aplicaciones web concurrentes, distribuidas, componibles. Esto se debe a que el framework está basado en una serie de principios funcionales que permiten representar estas funcionalidades en código de una manera muy sencilla.

El framework está divido en tres partes:

- Transient (core): Librería de manejo de concurrencia. Define un tipo `TransientIO` que permite componer aplicaciones (funciones basadas en IO) de una manera ilimitada. Junto con este tipo se definen otros operadores como `<|>` que permite ejecutar dos aplicaciones en paralelo y devolver el primer resultado satisfactorio o `<*>` que permite combinar el resultado de dos aplicaciones en una única (equivalente a *join*). Esto se debe gracias a que `TransientIO` es una estructura de datos que se adhiere a las clases de tipo `Applicative` y `Alternative`.

  Además, se permite la secuenciación de operaciones `TransientIO` de una manera imperativa gracias a que la estructura de datos de `TransientIO`

implementa la clase de tipo `Monad`.

- Transient Universe: Librería de computación distribuida. Se basa en las primitivas definidas por la librería core. Además define un tipo `Cloud` que permite ejecutar aplicaciones `TransientIO` en distintos nodos de ejecución por la red.

  Gracias a que `Cloud` implementa `Applicative`, `Alternative` y `Monad`, todas las reglas aplicables a `TransientIO` tambien lo son para `Cloud` (ejecución condicional, secuencial y *join*).

  Transient Universe también permite manejar nodos de ejecución y aplicar algoritmos de balanceo de carga.

- Axiom: Librería de generación de páginas web. La genialidad de Transient es que aprovecha la posibilidad de compilar Haskell a JavaScript y permite incrustar un nodo de ejecución en el navegador. Esto elimina completamente la necesidad de utilizar APIs REST (o SOAP), WebSockets o cualquier tipo de conexión explicita.

  Por supuesto, con ello viene la serialización y deserialización transparente, lo que significa que el desarrollador no se ha de preocupar por formatos como JSON o XML para el transporte. Transient administra todo esto.

  Además, Axiom define un eDSL, o embedded Domain Specific Language (Lenguaje incrustado Específico de Dominio) para desarrollar HTML, por lo tanto, tampoco nos hemos de preocupar por escribir HTML directamente y podemos utilizar funciones predefinidas por Axiom para esta tarea.

Por supuesto, no todo es de color de rosa, ya que Transient no posee una amplia comunidad, y menos, una amplia documentación. Para entenderlo se ha requerido indagar en el código fuente y contactar con Alberto Gómez.

5.4.2 Clay

Clay define un eDSL para escribir CSS dentro de Haskell. Esto ofrece la ventaja de que nos permite generar estilos personalizados basados en valores tanto en tiempo de ejecución como de compilación, además de que podemos utilizar cualquier tipo de función de Haskell.

5.4.3 QuickCheck

Para testing, se ha utilizado la librería QuickCheck, que permite hacer pruebas unitarias basadas en propiedades. Esto significa que en vez de asegurar que para ciertos valores la unidad que estamos testeando da el comportamiento correcto, definimos una propiedad y QuickCheck se asegura de inyectar valores aleatorios, intentando hacer que falle. Se han dado casos, en los que gracias a esto, se han descubierto bugs que eran impensables.

5.5 Otras herramientas utilizadas

Además de las anteriormente mencionadas, se han utilizado algunas herramientas de software adicionales:

5.5.1 Git

Git es un software de control de versiones.

5.5.2 GitHub

GitHub es una plataforma web pensada para alojar proyectos software, gestionar incidencias y fomentar la comunicación comunidad-desarrollador.

5.6 Requisitos de hardware

5.6.1 Un ordenador con GNU/Linux o OSX con conexión a Internet.

Dado que la mayoría de usuarios poseen conexión a Internet, se ha optado por que la aplicación descargue los assets necesarios tras su ejecución. También, a su vez, el asistente de creado de proyectos utiliza una plantilla que se encuentra online, de modo que si hay que resolver algún fallo, o añadir una característica inicial al proyecto, el usuario no ha de descargar una versión nueva de Haskell.do .

Capítulo 6

Metodología

6.1 ¿Qué es Scrum?

Scrum proviene del nombre de una formación del Rugby. Los jugadores se agazapan y se atenazan contra el equipo contrario, intentando empujar el balón fuera de la formación. En este sentido, no colaboran como personas individuales, sino como un todo, prevaleciendo los intereses del grupo.

Los investigadores Nonaka y Takeuchi se inspiraron en este concepto para dar pie en 1986 al uso de este término en el campo del desarrollo. Este término caracterizaba a los equipos que lo utilizaban como auto-organizados y motivados, buscando un entorno abierto tanto de trabajo como aprendizaje. El conocimiento se comparte en estos equipos y se brinda al grupo de la autonomía suficiente como para ser auto-responsables.

En 1995 nació formalmente Scrum en el campo del desarrollo del software. Dando lugar a un proceso iterativo que persigue un conjunto de buenas prácticas, centrándose en el proceso incremental. Gracias a esta priorización, se disminuye el tiempo que tarda un producto en salir al mercado y favorece que el software esté en constante cambio y mejora.

6.2 Aplicando Scrum

En Scrum, los proyectos se ejecutan en bloques. Estos bloques son cortos en cuanto a duración y fijos, denominándose sprints. Cada sprint proporciona un resultado completo, el cual está listo para ser entregado al cliente.

Cada bloque parte de la lista de requisitos establecidos por el cliente, denominado Product Backlog.

El proceso se define de la siguiente manera:

1. Se definen las funcionalidades globales del producto deseado escribiéndolas como historias de usuario: `Como <rol>quiero <descripción de functionalidad>para <razón de la funcionalidad>`
2. Se estiman las prioridades de las historias de usuario y se le asignan puntuaciones.
3. Se añaden al Product Backlog.
4. Se eligen las historias de usuario más prioritarias y se pasan del Product Backlog, al Sprint Backlog. Este es el listado de historias que se van a realizar durante el sprint.
5. Se realiza el sprint definido: a. Se desglosan las historias de usuario en tareas de desarrollo. b. Se definen los estados de las tareas que se representan en un tablón Kanban. c. Se va actualizando el tablón Kanban a medida que se va actualizando su estado.
6. Al final de cada sprint, se entrega el un incremento funcional. Se refina el Product Backlog y se repite el proceso

Además a este flujo hay que añadirle varias reuniones periódicas para controlar la evolución del proyecto:

- **Planificación del producto**. Reunión para que el cliente defina las funcionalidades del producto, abarcando todos los posibles objetivos.
- **Daily meeting**. Reunión diaria, de unos 15 minutos, donde los miembros del equipo discuten sobre lo realizado el día anterior, que problemas han tenido y que se espera realizar ese día.
- **Sprint review**. Reunión tras la entrega del sprint, en la que el cliente ve y comprueba el avance funcional de su producto tras el sprint realizado.
- **Refinado de la pila de producto**. Reunión tras la entrega del sprint, donde se refinan las historias actuales, se añaden nuevas, se replantean las prioritarias y se planifica el siguiente sprint.

6.3 Scrum en Haskell.do

A pesar de que Scrum es una metodología de gestión para equipos, también se encuentra útil para gestionar proyectos realizados por un solo individuo, ya que esta metodologías está compuesta por varias prácticas muy útiles.

6.3.1 Historias de usuario

A diferencia de los procesos de la ingeniería del software tradicional, donde se elaboran los requisitos mediante diagramas, tanto de casos de uso, como de actividad o flujo, Scrum apoya la recogida de requisitos en un único elemento, las **historias de usuario**.

En cada historia de usuario se detallan varios aspectos:

- La redacción de la historia con el formato explicado anteriormente.
- La descripción de los escenarios para poner en contexto.
- Tareas asociadas a cada historia de usuario.
- Puntuación de dificultad de la historia de usuario.

6.3.2 Product backlog

Todas las historias se añaden al product backlog en orden de prioridad. De este product backlog se extraen historias para resolver durante el sprint.

6.3.3 Sprints

El total del proyecto se ha descompuesto en tres sprints, existiendo un producto funcional que se acerca cada vez más al producto final deseado.

Estos sprints, así como cada uno de sus backlogs, serán detallados más adelante en sus apartados correspondientes.

6.3.4 Tablón

A cada sprint le corresponde un tablón donde se podrá ver a alto nivel el estado actual de las historias de usuario y sus correspondientes tareas.

6.3.5 Gráfica burndown

Se puede observar el progreso del sprint gracias a esta gráfica, ya que nos permite ver si estamos cumpliendo con los tiempos necesarios para completar los puntos del sprint antes de llegar a la fecha de entrega.

6.4 ¿Qué es Git?

Git es un software de control de versiones diseñado por Linus Torvalds. Al principio Git se pensó como un motor donde otros pudieran escribir la interfaz de usuario o front-end. Sin embargo, se ha convertido desde entonces en un sistema de control de versiones con funcionalidad plena.

Entre sus características más importantes se encuentran:

- Apoyo al desarrollo no lineal, gestionando ramas y diferentes versiones. Incluye herramientas de navegación y visualización de todas las versiones y ramas. La premisa de Git es que un cambio se fusione más frecuentemente de lo que se escribe originalmente.
- Gestión distribuida, donde cada programador tiene una copia local del historial del desarrollo entero, y los cambios se propagan entre los repositorios locales. Los cambios se importan como ramas adicionales y se pueden fusionar de la misma manera que se hace con la rama local.
- Gestión eficiente de proyectos a gran escala. Git provee de una notoria velocidad en la gestión de diferencias entre archivos.
- Todas las versiones previas a un cambio determinado implican la notificación de un cambio posterior en cualquiera de ellas a ese cambio.
- Los renombrados de fichero se trabaja en base a similitudes entre ficheros.

6.5 Git Flow

Git Flow es un flujo de trabajo basado en ramas, utilizado por muchísimas organizaciones y empresas por todo el mundo. Ofrece una manera directa de trabajar con nuestro código a la hora de implementar el proyecto, evitando así, los conflictos con otros miembros del equipo.

6.5.1 Rama Develop

En Git Flow, en vez de escribir todos nuestros cambios de desarrollo sobre la rama principal, lo hacemos sobre la rama `develop`, dejando la principal, `master`, solo para despliegues y versiones estables.

Esto nos permite asegurarnos bien de que nuestros cambios son testeados profundamente, antes de enviarle al cliente una versión.

6.5.2 Features

Toda adición al proyecto se considera característica. Por lo tanto, cuando vayamos a añadir algo al proyecto, crearemos una rama con el prefijo `feature/`. De esta manera, los desarrolladores que trabajen en características distintas, no tienen porque pisarse.

Al finalizar la característica, hacemos una petición de revisión al resto del equipo, y si lo ven correcto, se fusionarán nuestros cambios con la rama `develop`.

6.5.3 Release

Al acumular suficientes características como para terminar un incremento, se genera una release. A la release se le asigna un código, generalmente utilizando versionado semántico[1] y se fusiona con la rama `release`.

En esta rama se arreglan bugs menores y finalmente se fusiona con la rama principal cuando sea necesario.

6.5.4 Hotfixes

Un hotfix es un *arreglo en caliente*, es decir, cuando ya hemos entregado/desplegado la versión de producción, pero nos hemos encontrado con un bug que ha de ser solucionado contra la versión en producción.

[1] El versionado semántico, también conocido como **semver**, es una especificación que indica los cambios que se han añadido a algo utilizando tres números separados por puntos, el primero para cambios mayores, que rompan retro-compatibilidad. El segundo para cambios menores, que añadan características a la versión actual. El tercero para arreglos de fallos que puedan ocurrir: **X.Y.Z**

Para ello, **desde la rama principal** se genera una rama con el prefijo `hotfix/` para arreglar este bug y finalmente, volver a fusionarla con la principal.

Capítulo 7

Planificación

7.1 Creación de la pila de producto: 20 horas.

Redacción de las Historias de usuario para la extracción de los requisitos funcionales.

Realmente, se ha tardado mucho menos, ya que los requisitos estaban bastante claros desde el principio, es por ello por lo que se ha tardado unas **10 horas**.

7.2 Estudio, diseño, implementación y pruebas: 200 horas.

Configuración de Emacs y del sistema operativo para desarrollo para Haskell. Familiarización con Transient, Clay y las demás tecnologías. Diseño de arquitectura. Implementación y finalmente testing.

Este proceso se dividirá en tres sprints:

- Sprint 1 : 80 horas
- Sprint 2 : 60 horas
- Sprint 3 : 60 horas

La duración de la parte de desarrollo ha sido exactamente como se había estimado, no ha habido desviación.

7.3 Documentación: 80 horas.

La redacción del manual de uso de la aplicación, la memoria del TFG y una presentación.

Realmente aqui se ha tardado 95 horas, ya que no se tenía tanta experiencia con LaTeX, han habido problemas con las referencias a las figuras y otros relacionados con elementos flotantes.

7.4 Desarrollo del Backlog

La parte más importante de un proyecto gestionado con Scrum, como se ha mencionado anteriormente, es el Backlog. Es la base de toda la organización de los futuros sprints. El backlog inicial es el que se puede observar en la Sección 11.1

Capítulo 8

Desarrollo del proyecto

8.1 Sprint 1

Tras definir el Product Backlog, se ha fijado que el primer sprint ha de tener una duración de 80 horas. Estas horas se han asignado como puntos, es decir, este sprint está compuesto de 80 puntos.

El Sprint Backlog correspondiente sería:

- Project Setup
- Ejecutar desde línea de comandos
- Distribución UI - Core (incluye diseño y desarrollo de UI)
- Menú de aplicación

8.1.1 Project setup

1. **Objetivo:** En bastantes proyectos gestionados por Scrum se discute si esta historia de usuario se ha de añadir a la pila de producto o no, ya que no concierne al cliente, sino al desarrollador. Es una historia de usuario crucial para el proyecto, ya que es la base del resto del desarrollo.

   El objetivo principal es preparar el entorno de trabajo, configurando los frameworks y herramientas necesarias para el desarrollo del proyecto

2. **Estimación:** 5 horas.

3. **Historia de usuario:** Como desarrollador, quiero configurar el entorno

de trabajo del proyecto, para poder trabajar correctamente.

4. **Escenario:** No tiene ningún escenario definido como tal, ya que es una tarea que concierne a los desarrolladores.
5. **Trabajo realizado:**
   - Instalación de GHC (compilador de Haskell) y GHCJS (compilador de Haskell a JavaScript).
   - Inicialización de un proyecto Haskell utilizando Stack y Git.
   - Alojamiento del proyecto en GitHub
   - Se han añadido las dependencias correctas al proyecto, como las librerías de Transient.
   - Generación de un test básico de propiedad, que realmente no testea nada.
   - Creación de un script en Haskell para construir el proyecto más rápidamente. Este script ejecuta la compilación de la parte de servidor y cliente (aún no se ha hecho distinción) y posteriormente empaqueta en una carpeta la aplicación compilada para poder ejecutarse.
6. **Tiempo empleado:** 5 horas

8.1.2 Ejecutar desde línea de comandos

1. **Objetivo:** El objetivo principal es permitir ejecutar la aplicación desde la linea de comandos para poder inicializarla desde cualquier sistema, tenga interfaz gráfica instalada o no. Junto con esto, se va a inicializar Transient en modo mono-nodo. El grueso de esta tarea es investigación para entender como funciona la inicialización de Transient.
2. **Estimación:** 5 horas
3. **Historia de usuario:** Como usuario, quiero iniciar la Haskell.do desde la terminal, para poder acceder desde cualquier sistema *NIX.
4. **Escenario:**
   - Me encuentro en la terminal
   - Escribo `haskell-do`

- Se inicia Haskell.do correctamente, indicándomelo con un mensaje.

5. **Trabajo realizado:**
   - Investigación de inicialización de Transient mono-nodo.
   - Escritura de código de inicialización
   - Creación de mensaje de bienvenida
6. **Tiempo empleado:** 5 horas

8.1.3 Distribución UI - core

1. **Objetivo:** Permitir al usuario acceder a una interfaz web completamente desacoplada del servidor de ejecución, permitiendo que la interfaz sea mucho más fluida que ejecutándola en el mismo sistema.

   Se ha de hacer el desacoplamiento y la interfaz gráfica en conjunto, ya que con el modelo de computación que sigue Transient no tiene sentido hacer esta parte por separado.
2. **Estimación:** 50 horas
3. **Historia de usuario:** Como usuario, quiero poder acceder a una dirección web que me provea una interfaz web que se ejecute del lado del cliente, para poder trabajar con Haskell.do fluidamente.
4. **Escenario:** Accedo a la dirección provista por el Haskell.do en el mensaje de bienvenida, o cualquier redirección a ella, me encuentro con una interfaz gráfica con forma de documento/hoja de papel.
5. **Trabajo realizado:**
   - Investigación acerca de la distribución nativo-navegador de Transient
   - Diseño de interfaz
   - Creación de interfaz con Transient utilizando Axiom.
6. **Tiempo empleado:** 65 horas.

   Se ha requerido crear un micro framework para gestionar los eventos de la UI reactiva y declarativamente, ya que Axiom está orientado a un uso más clásico.

   Además, la conexión de librerías de JavaScript con la interfaz escrita en Haskell no ha sido trivial, ya que JavaScript no es un lenguaje tipado

y Haskell si lo es, por lo tanto requiere que se declaren y este hecho no estaba documentado en ningún sitio a la vista.

8.1.4 Menú de aplicación

1. **Objetivo:** Proveer al usuario de una manera sencilla de acceder a las funciones básicas de la aplicación:
   - Crear proyecto
   - Abrir proyecto
   - Configurar el proyecto
   - Gestionar las dependencias del proyecto
   - etc...
2. **Estimación:** 10 horas
3. **Historia de usuario:** Como usuario, quiero poder acceder fácilmente a las características básicas de la aplicación, para así poder ejecutar las funcionalidades correspondientes.
4. **Escenario:** Estoy en la interfaz principal, accedo al menú y:
   - Clico en nuevo proyecto, me aparece la vista de creación de proyecto.
   - Clico en abrir proyecto, me aparece la vista de abrir proyecto.
   - Clico en configurar proyecto, me aparece la vista de abrir proyecto.
5. **Trabajo realizado:** Se ha generado un menú encapsulado en un botón flotante, estilo *material design* [1].

   Se han añadido tres botones, con su correspondiente generación de eventos:
   - Crear o Abrir proyecto (se utilizará la misma vista).
   - Configuración de proyecto (se utilizará la misma vista).
   - Ver/Ocultar editor de texto. Esto es útil por si el usuario quiere solo visualizar un código y su resultado existente.
6. **Tiempo empleado:** 10 horas.

[1] Material design es un estándar de diseño gráfico diseñado por Google.

8.1.5 Gráfica de sprint

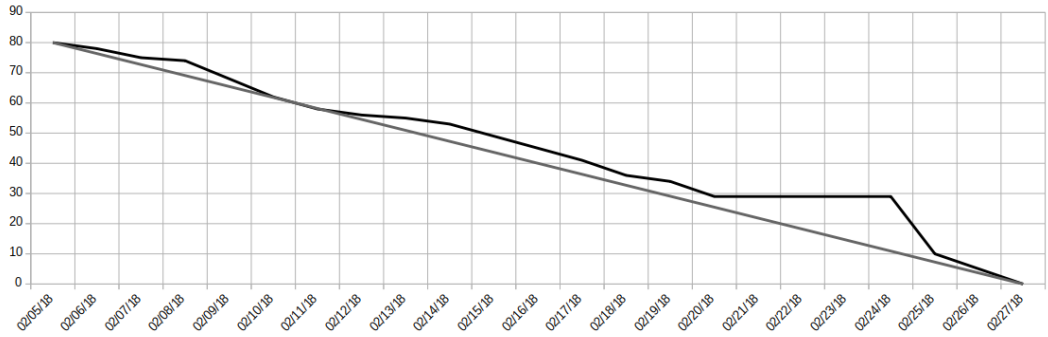

Figura 8.1: Burndown chart del sprint 1

En esta gráfica se puede ver como la tarea de "distribución UI-core"ha influido negativamente en el sprint, teniendo que hacer un sobre-esfuerzo para conseguir llegar finalmente a la deadline.

En la estimación del sprint se tuvo una opinión demasiado optimista, pero se ha ido avanzando correctamente hasta llegar a la tarea que se ha mencionado anteriormente. Se ha decidido invertir horas de más en esta tarea, para poder cumplir con las expectativas.

8.1.6 Incremento funcional

Figura 8.2: Incremento funcional del sprint 1

En la Figura 8.2 podemos ver como se ha creado la interfaz básica, con forma de documento. Arriba a la derecha, se encuentra el menú de aplicación desplegable.

8.1.7 Conclusiones

En este primer sprint se ha realizado muchísima investigación y "preparación del terreno", puesto que a pesar de conocer el lenguaje, las tecnologías utilizadas, como Transient o GHCJS, no son familiares.

La documentación de Transient es mucho más deficiente de lo que se esperaba. A pesar de no ser demasiado complejo de utilizar, nadar en el código fuente es bastante complejo ya que no sigue las guías de estilo de Haskell, ni tampoco los patrones de diseño predefinidos. Sin embargo, el esfuerzo vale la pena, pues el tiempo de desarrollo se reduce muchísimo.

GHCJS es un invento formidable, pues compila Haskell a JavaScript con todas las funcionalidades del runtime de Haskell como multithreading, evaluación perezosa y capacidad para utilizar extensiones del lenguaje.

8.2 Sprint 2

Tras finalizar el primer sprint, no hay que refinar el Product Backlog, puesto que todas las tareas del sprint anterior se realizaron correctamente.

El Sprint Backlog correspondiente es:

- Creación de un proyecto
- Apertura de un proyecto existente
- Configuración del proyecto
- Gestión de dependencias

8.2.1 Creación de un proyecto

1. **Objetivo:** El objetivo de esta historia es ofrecer al usuario una vista que permita inicializar un proyecto de Haskell.do en el sistema de archivos del servidor. Ojo, no es en el lado del cliente, pues la ejecución ocurre en el lado del servidor.

2. **Estimación:** 35 horas
3. **Historia de usuario:** Como usuario, quiero poder crear un proyecto nuevo desde la aplicación, para poder inicializar un nuevo documento donde ejecutar el código interactivamente.
4. **Escenario:** Me encuentro en la vista principal, abro el menú de aplicación, hago clic en crear proyecto, la vista de proyecto aparece, creo el proyecto con la ayuda de la UI.
5. **Trabajo realizado:** Al principio del desarrollo de esta historia, se había optado por generar la estructura del proyecto en base a un asistente de creación, el cual preguntaba al usuario parámetros básicos como:

   - Nombre del proyecto
   - Versión
   - Sitio web del proyecto
   - etc...

   Pero esto no solo se volvía superfluo para el usuario (esto es configurable en cualquier momento), sino que desarrollar la interfaz estaba costando más de lo estimado.

   Finalmente, se desarrolló un pequeño explorador de archivos que permite elegir una carpeta, y más adelante podría ser reutilizado en la historia de abrir proyecto.

   Se decidió crear una template de Stack y alojarla en GitHub, para que fuera Stack el encargado de inicializar un proyecto base, configurable más adelante.

   Haskell.do, tras esperar a que Stack creara el proyecto y descargara las dependencias necesarias, carga los datos del proyecto en memoria y muestra el código del proyecto en el editor, el cual ha sido implementado en esta tarea.
6. **Tiempo empleado:** 35 horas.

8.2.2 Apertura de un proyecto existente

1. **Objetivo:** Permitir al usuario abrir un proyecto existente creado en Haskell.do.

2. **Estimación:** 5 horas.
3. **Historia de usuario:** Como usuario, quiero poder abrir un proyecto creado anteriormente, para poder continuar mi trabajo.
4. **Escenario:** Estoy en la pantalla principal, abro el menú de aplicación, clico en el botón de nuevo/abrir, abro el proyecto con la ayuda de la UI.
5. **Trabajo realizado:** Se ha reutilizado casi todo lo realizado en la historia anterior, con la excepción de que se ha añadido una advertencia al usuario para decir que si abre una carpeta donde no existe un proyecto se creará uno. Además se ha desacoplado el código de carga de datos de la parte de creación del proyecto para que pudieran usarlo ambos módulos.

   Se han hecho los cambios pertinentes en la UI.
6. **Tiempo empleado:** 5 horas.

8.2.3 Configuración del proyecto

1. **Objetivo:** Permitir al usuario configurar su proyecto correctamente con los valores apropiados de:
   - Nombre
   - Versión
   - Repositorio GitHub
   - Categoría
   - Sitio web
2. **Estimación:** 10 horas
3. **Historia de usuario:** Como usuario, quiero poder configurar mi proyecto, para así poder distribuirlo en el repositorio de paquetes de Haskell.
4. **Escenario:** Me encuentro en la pantalla principal, abro el menú de aplicación, clico en el botón de ajustes, configuro el proyecto con la ayuda de la UI.
5. **Trabajo realizado:** Al principio se pretendía que la interfaz ofreciera un formulario clave-valor para todas las opciones de configuración del

proyecto, pero puesto que eran muchas más de las esperadas, con formatos diversos para el valor (string, numeros, listas enumeradas...), se había comenzado a desarrollar esto con todas las opciones posibles mostradas en pantalla.

Luego, se vio que esto hacía que la vista fuera engorrosa y se comenzó a desarrollar un "buscador"que auto-sugería valores y permitía ir añadiendo valores dinámicamente al formulario, donde se invirtió muchísimo más tiempo del que se debía, superando la estimación con creces.

Como solución fácil y rápida, que además contaba con la ventaja de que toda la comunidad Haskell la conocía, se añadió un editor de texto secundario que editaba el fichero de configuración directamente, solucionando a la vez la siguiente historia de usuario.

6. **Tiempo empleado:** 20 horas.

8.2.4 Gestión de dependencias

1. **Objetivo:** Permitir al usuario añadir y quitar librerías a su proyecto.
2. **Estimación:** 10 horas.
3. **Historia de usuario:** Como usuario, quiero poder añadir dependencias a mi proyecto, para poder utilizar librerías adicionales, no quedándome con lo básico del lenguaje.
4. **Escenario:** Estoy en la pantalla principal, abro el menú de aplicación, clico en configurar proyecto, añado dependencias con la ayuda de la UI.
5. **Trabajo realizado:** No se ha tenido que realizar ningún trabajo, puesto que la solución a la historia anterior ha solucionado también esta.
6. **Tiempo empleado:** Cero.

8.2.5 Gráfica de sprint

Como podemos ver en esta gráfica, la progresión ha sido satisfactoria, ya que se ha acercado al progreso ideal. Por otro lado, se observa como la tarea de Configuración del proyecto ha estancado el progreso, finalizando el sprint satisfactoriamente, solo porque se ha cumplido indirectamente la tarea de gestión de dependencias.

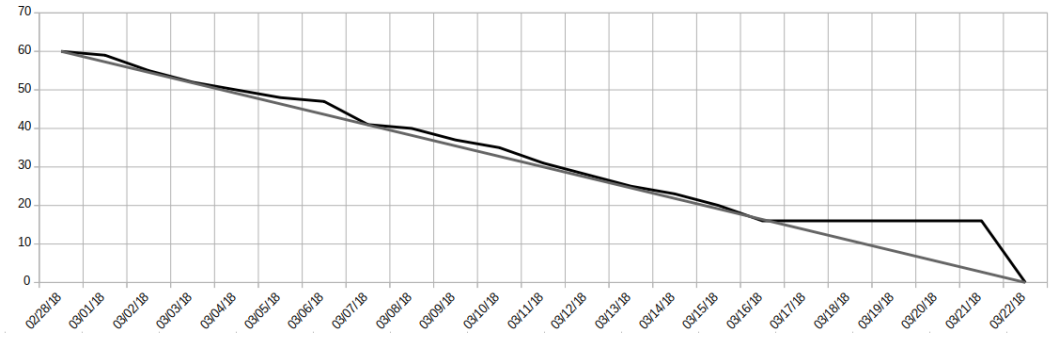

Figura 8.3: Burndown chart del sprint 2

8.2.6 Incremento funcional

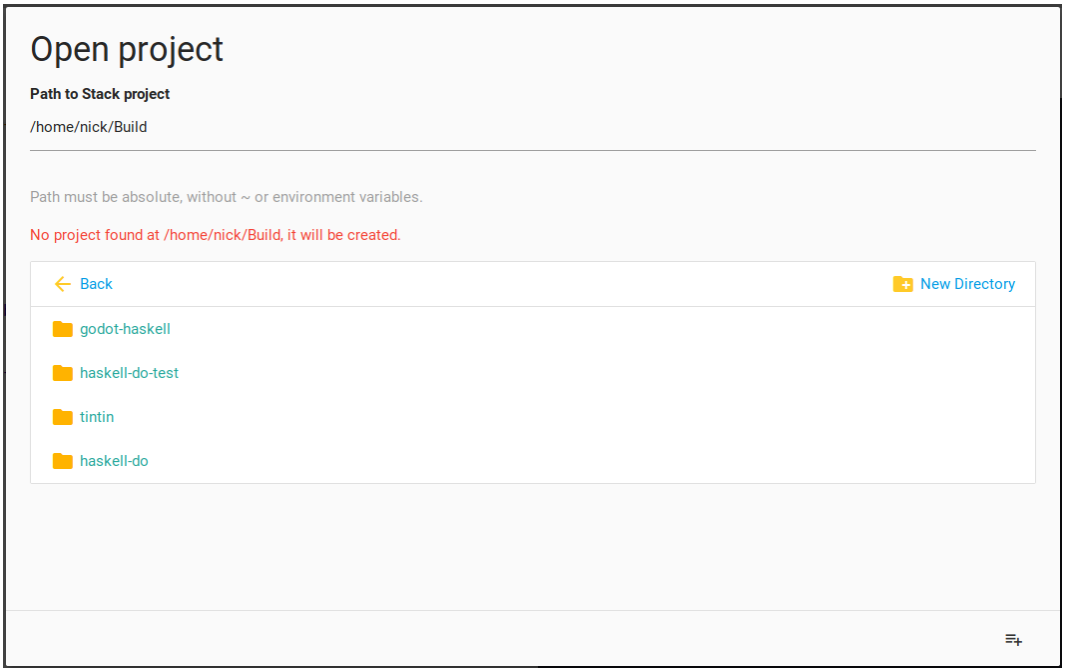

Figura 8.4: Vista de crear/abrir proyecto

En la Figura 8.4 podemos observar el dialogo de creación y apertura del proyecto, mientras que en la Figura 8.5 podemos observar el editor básico para modificar la configuración del proyecto, junto con las dependencias.

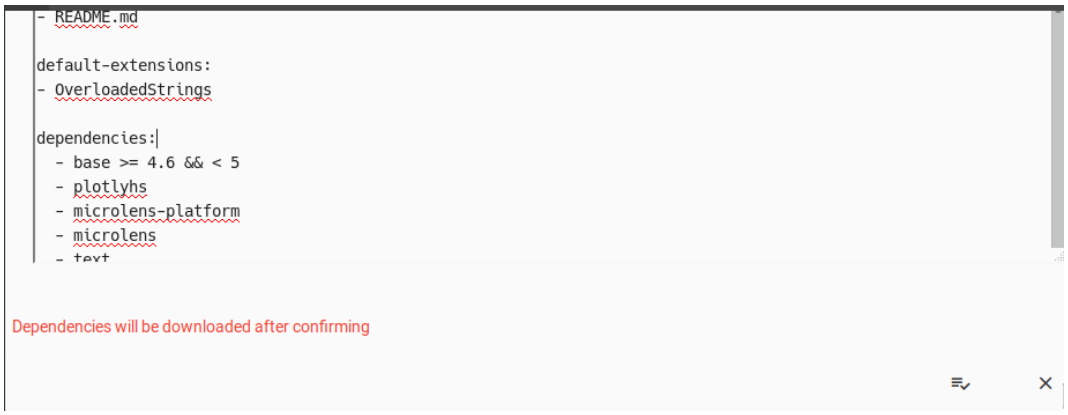

Figura 8.5: Vista de configuración/dependencias

8.2.7 Conclusiones

Este sprint ha sido dual en cierta manera: Por un lado, ha sido muy satisfactorio, puesto se ha avanzado en cuanto a las características esperadas, pero por otro lado las estimaciones apenas han servido de nada, ya que se han optado por soluciones "dos en uno".

Sin embargo, este sprint ha incentivado la motivación, puesto que se han empezado a ver características útiles de la aplicación.

En cuanto a aprendizaje, me ha ayudado a:

- Aprender como definir creación de UI en runtime de una manera *type-safe* [2].
- Aprender a interaccionar con procesos del sistema con Haskell.
- Maquetar correctamente un explorador de archivos.

8.3 Sprint 3

Tras finalizar el segundo sprint, no se ha necesitado refinar el Product Backlog una vez más.

Se añaden el resto de historias al Sprint Backlog:

- Evaluación de código Haskell como IO

[2]El type safety es un tipo de verificación formal, donde el compilador comprueba de

- Definición de funciones en ámbito global
- Documentar el código con Markdown
- Generación de gráficas interactivas

8.3.1 Evaluación de código Haskell como IO

1. **Objetivo:** Permitir al usuario ejecutar el código de una manera secuencial, como si se tratase de la función `main`, que se ejecuta dentro de una `Monad`.
2. **Estimación:** 15 horas.
3. **Historia de usuario:** Como usuario, quiero ser capaz de ejecutar código interactivamente, para poder ver los resultados instantáneamente.
4. **Escenario:** Estoy en la pantalla principal, escribo en el editor de código (izquierda), presiono `CTRL + Enter`, el resultado aparece a la izquierda.
5. **Trabajo realizado:** Se ha añadido editor más complejo, puesto que antes se utilizaba un text-area simple provisto por HTML5, pivotando ahora a CodeMirror, que ofrece resaltado de sintaxis.

   El código escrito a la izquierda, de una manera secuencial, en la `Monad IO`, se manda a un proceso interactivo de GHCi (compilador de Haskell en modo interprete).

   El resultado se muestra a la izquierda tras ser evaluado. Tanto los satisfactorios, como los fallidos.
6. **Tiempo empleado:** 15 horas.

8.3.2 Definición de funciones en ámbito global

1. **Objetivo:** Hasta ahora, el usuario solo podía ejecutar código secuencial en la `Monad IO`. De modo que, si este quisiera poder definir una función, tendría que especificarlo de la misma manera que lo haría secuencialmente:

   ```
   let f = (\x -> x * 2) :: Int -> Int
   ```

   Esta definición, no es lo estándar, y se considera una mala práctica de

antemano todas las posibilidades, y advierte de posibles casos de error.

Haskell, puesto que, claramente, se ha de refactorizar como una función en ámbito global, de la forma:

```
f :: Int -> Int
f x = x * 2
```

O incluso:

```
f :: Int -> Int
f = (* 2)
```

El objetivo de esta historia de usuario es permitir al usuario escribir código de esta manera.

2. **Estimación:** 15 horas
3. **Historia de usuario:** Como usuario, quiero poder escribir funciones en el ámbito global, para poder organizar las funcionalidades de mi proyecto correctamente.
4. **Escenario:** Estoy en la pantalla principal, escribo código secuencial y código de alto nivel, al presionar `CTRL + Return` el resultado aparece a la izquierda.
5. **Trabajo realizado:** Puesto que GHCi no permite declaraciones de ámbito global línea a línea, se ha optado por parsear las líneas que comenzaran con una palabra, un espacio y un doble dos puntos, indicando el comienzo de una declaración de función.

   Las lineas siguientes no vacías se concatenarían con un punto y coma en vez (`;`) en vez de un salto de línea, permitiendo añadirlo a GHCi.

   Rápidamente, esto se volvió inestable, puesto que si alguien escribía una función tal que:

   ```
   f :: Int -> Int
   f x =
       x * y + c
     where
       y = 4

       c = 2
   ```

   `c` seguía perteneciendo al ámbito de `f`, pero el algoritmo utilizado lo declaraba como constante global.

   Se descubrió que GHCi permite entrada multilínea delimitando con

`:{` y `:}`. De modo que se hizo que Haskell.do parseara las líneas con comentarios especiales, y los convirtiera en delimitadores:

```
-- TOP_DECL start
f :: Int -> Int
f x =
    x * y + c
  where
    y = 4

    c = 2
-- TOP_DECL end
```

6. **Tiempo empleado:** 15 horas

8.3.3 Documentación del código en Markdown

1. **Objetivo:** Editores como R Studio y Jupyter permiten anotar el código con Markdown, ayudando a la documentación de este.

   El código escrito en estos editores a menudo suele representar fórmulas matemáticas, algoritmos de machine learning y parecidos, de modo que no suele ser trivial entenderlo a la primera. Markdown es una ayuda importante en estos casos, ya que permite formatear los comentarios automáticamente.

   El objetivo de esta historia de usuario es permitir que el usuario pueda escribir código Markdown con sus correspondientes bloques de código Haskell, y que Haskell.do pueda interpretarlos correctamente, de esta manera a la izquierda aparecería un documento renderizado con los resultados correspondientes.

2. **Estimación:** 25 horas

3. **Historia de usuario:** Como usuario, quiero poder escribir documentación Markdown *inline*, para poder generar documentación con formato enriquecido automáticamente para mi código.

4. **Escenario:** Estoy en la pantalla principal, escribo código Markdown con bloques Haskell intercalados en el editor a la izquierda, presiono `CTRL + Enter` y el resultado aparece a la derecha.

5. **Trabajo realizado:** Se escribió un algoritmo que permitía enviar las líneas escritas en bloques de código Haskell a GHCi, ya que trataba

todas las restantes como comentarios.

Para el formato de la documentación se hizo uso de la librería `cheapskate`, finalmente se inyectaba el resultado al final del documento como `Result:`.

El problema de esto es que no es realista, ya que un código de exploración para ciencia de datos, nunca tiene una única salida, sino muchas. Se comenzó una discusión en el canal de Gitter de dataHaskell con la comunidad para averiguar una posible solución.

Uno de los miembros resaltó la existencia de la librería `inliterate`, que permite compilar Markdown como si fuera código Haskell, ya que se hacía un preprocesado previo.

Esto llevaba dos consecuencias duales:

- **La mala:** El código pasaba a ser compilado, por lo tanto se añadía un retardo a la ejecución, la cual ya no era instantánea.
- **La buena:** Se podían activar extensiones de lenguaje, las cuales no eran posibles anteriormente, puesto que funcionan en el compilador y no el intérprete. Además, ya que el código era compilado, era mucho más rápido.

Sin lugar a duda, teniendo en cuenta que solucionaba el problema, se cambió la creación de proyecto para que dependiera de `inliterate` y se hizo la integración.

Esto vino genial, ya que se había excedido la *deadline*.

6. **Tiempo empleado:** 30 horas.

8.3.4 Generación de gráficas

1. **Objetivo:** En los editores interactivos, es importante que se puedan generar gráficas para representar resultados de algoritmos o cualquier otra cosa.

   Con Haskell.do se quería ir más allá y hacer una integración con alguna librería de JavaScript donde las gráficas fueran interactivas de alguna manera, como por ejemplo, hacer zoom y poder desplazarte.

   El objetivo de esta historia es implementar esta característica.

2. **Estimación:** 5 horas.

3. **Historia de usuario:** Como usuario, quiero poder dibujar gráficas interactivas, para poder sacar conclusiones de mis análisis de una manera sencilla.

4. **Escenario:** Estoy en la pantalla principal, escribo código que permita dibujar una gráfica en el editor, presiono `CTRL + Enter` y aparece una gráfica a la derecha.

5. **Trabajo realizado:** Mientras se trabajaba con `inliterate`, se descubrió que el autor poseía unos *bindings* a `plotly.js`, una de las mejores librerías de gráficas de JavaScript.

   Lo bueno de esto, es que estos bindings permiten representar la gráfica como una etiqueta HTML, que si se devuelve como resultado en un bloque Haskell, se pinta automáticamente en el documento renderizado.

6. **Tiempo empleado:** Cero.

8.3.5 Gráfica de sprint

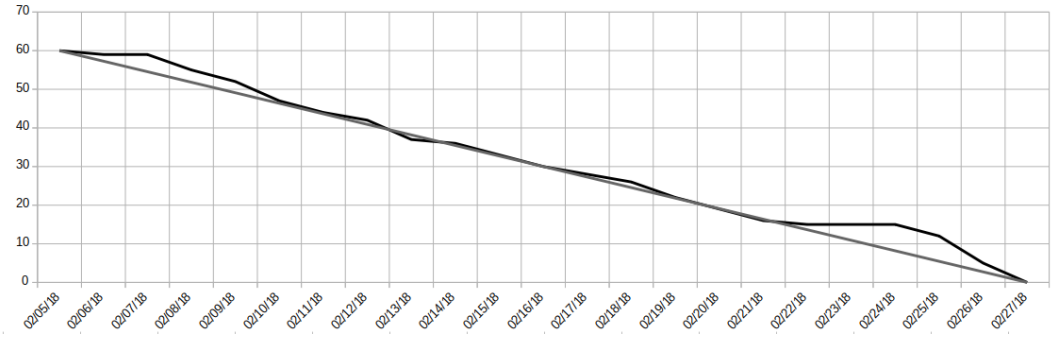

Figura 8.6: Burndown chart del sprint 3

Este sprint ha sido absolutamente satisfactorio. Se ha tenido un pequeño percance con la tarea de Documentación de código con Markdown, pero finalmente, todo el progreso ha sido perfecto, ya que en la tarea de Generación de gráficas no se ha tenido que realizar nada, puesto que las herramientas utilizadas para la tarea anterior ya suplían esta necesidad.

8.3.6 Incremento funcional

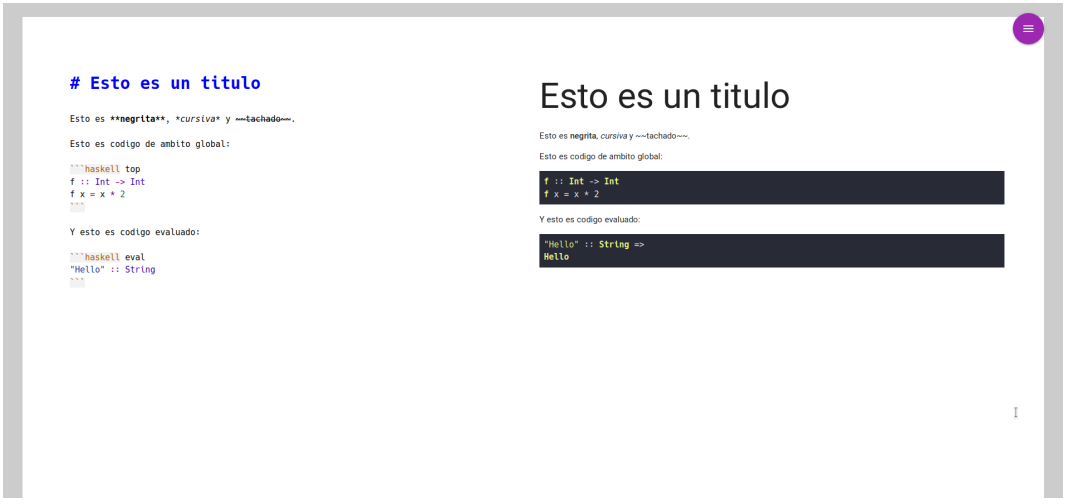

Figura 8.7: Documento Markdown y código

En la Figura 8.7 podemos ver como se renderiza a la izquierda el código Markdown, además de evaluarse Haskell. Por otro lado, en la Figura 8.8 se

Figura 8.8: Generación de gráfica

puede ver como se genera una gráfica interactiva de Plot.ly.

8.3.7 Conclusiones

Terminando este sprint, se termina la aplicación. Al igual que en el sprint anterior, resultó que existían herramientas existentes para las tareas que se pretendían. El único problema era integrarlas.

Las funcionalidades esperadas se quedan cubiertas, dejando una aplicación funcional que se puede utilizar para programación interactiva.

En cuanto a aprendizaje:

- Se aprendió a parsear utilizando `megaparsec`, una librería de parsing basada en *parser-combinators*, una alternativa al clasico LALR, planteada de manera funcional.
- Se aprendió a utilizar la librería `inliterate` para generación de documentos con Haskell
- Se aprendió a utilizar PlotlyJS para especificar gráficas.
- Llevar la organización el proyecto utilizando Scrum.

Capítulo 9

Presupuesto

Siendo un proyecto open source, sin ánimo de lucro, no se ha buscado una rentabilidad, por lo tanto el presupuesto no ha sido un factor importante en el proyecto. Sin embargo, en cuanto a costes, el cálculo se reduciría a:

Categoría	**Descripción**	**Coste**
Equipo	Sistema donde se desarrolló el proyecto. Thinkpad X250 semi-nuevo.	1950€
Dominio	Dominio empleado para la página web del proyecto. Importante para el marketing.	360€
Horas de trabajo	Coste hora × Horas empleadas en el desarrollo del proyecto (12.50€/hora)	
	Horas Arquitecto - 40	1500€
	Horas Analista - 30	1125€
	Horas Diseñador - 10	375€
	Horas Programador - 120	4500€
	Total	**9810€**

Capítulo 10

Conclusiones y trabajo futuro

10.1 Conclusiones

Quería resaltar, antes que nada, que estoy muy orgulloso del trabajo que se ha hecho. El proyecto se ha presentado en Suiza (Zurihac, 2017) y varias conferencias de España (LambdaWorld, 2016 y HaskellMAD, 2017), con una muy buena opinión del público. Además, a pesar de que es una aplicación que aún se puede mejorar para que sea estable y robusta, es mucho mejor que las alternativas propuestas en el entorno de la ciencia de datos de Haskell.

Alberto Gómez Corona, el creador de Transient, ha sido de muchísima ayuda durante el desarrollo de este proyecto, y gracias a él, y su genial framework, ha sido posible acabarlo en el tiempo estimado.

Por supuesto, que no todo el mérito ha sido de Alberto, pues la mayoría del tiempo he tenido que "nadar"por el código fuente (el cual está mal documentado y no sigue para nada las buenas prácticas de código limpio) y aprender a utilizar los distintos patrones de distribución que ofrece.

Mi formación ha avanzado muchísimo en cuando a auto-organización. A pesar de que conté con Nayara Rodríguez, project manager de Theam, para que me apoyara en ella si tenía dudas con respecto a Scrum, la formación obtenida en la Escuela de Ingeniería Informática de la ULPGC me ha permitido realizar esto a mi solo. Al enfocar el proyecto de esta manera, he sentido la presión de las deadlines de las estimaciones y me ha forzado a sacar features hacia delante. Si lo hubiera hecho de otra manera, no hubiera conseguido avanzar en la vida.

El hecho de que este haya sido mi primer proyecto Haskell de esta magnitud, me ha ayudado a indagar más en este maravilloso lenguaje que no se nombra en la carrera y en su patrones de diseño. Este proyecto me ha dado la oportunidad de entender como aplicar concurrencia y programación distribuida a gran escala, además de prevenir errores de runtime utilizando técnicas para hacer el desarrollo más *type-safe*.

La comunidad de dataHaskell también ha jugado un papel importante en el desarrollo de Haskell.do, permitiéndome tomar un poco el rol de *community-manager* para poder saber que características de verdad necesitaba la comunidad y cuales eran superfluas.

Para terminar, quería comentar que he obtenido un apoyo increíble por parte de Theam con este proyecto, tanto moralmente como económicamente, puesto que los proyectos open source a menudo se mueren sin estos dos factores. En el futuro se volverá a retomar el desarrollo de Haskell.do para convertirlo en, posiblemente, el IDE estándar para ciencia de datos en Haskell.

10.2 Trabajo futuro

A pesar de que el trabajo ha dado fruto a un proyecto satisfactorio, usable y funcional, existen bastantes posibilidades de mejora para hacer el proyecto aun mejor, y preparado para el mundo real:

- Lo más importante, cambiar el modelo de evaluación y desechar `inliterate`. Tras haber terminado el proyecto y haberlo probado varios usuarios, se han encontrado con el problema de que si algún bloque de código es pesado, como por ejemplo el calculo de un producto matricial con matrices enormes, ralentiza la ejecución y además se ha de ejecutar cada vez que se re-evalúe el documento. Haciendo en estos casos el programa inutilizable.
- Cambiar la interfaz de configuración del proyecto, haciendo una ventana con los campos necesarios.
- Añadir un buscador de dependencias, que permita buscar en el repositorio de paquetes de Haskell, y añadirlas automáticamente.
- Resaltado de errores en el código. Actualmente se muestra un dialogo rojo mostrando el error en bruto, pero es mucho mejor que se resalte la línea correspondiente.

- Etiquetado de las líneas. Actualmente cuando se hace la conversión Markdown a Haskell, el número de las líneas originales se pierden, y hace que los errores no contengan el número de línea correcto.
- Formato del Markdown en tiempo real, permitiendo que el usuario vea el texto enriquecido según va escribiendo.
- Una aplicación de escritorio cliente que se pueda abrir desde un acceso directo.

Capítulo 11

Anexo

11.1 Lista de funcionalidades

- Project setup
- Ejecutar desde linea de comandos
- Distribución UI - core de ejecución
- Menú de aplicación
- Creación de proyecto
- Apertura de proyecto existente
- Configuración del proyecto
- Gestión de dependencias
- Evaluación de código Haskell como IO
- Definición de funciones a alto nivel
- Documentación en Markdown
- Generación de gráficas

11.2 Detalles técnicos

Se ha de destacar que gracias al lenguaje de programación elegido el proyecto ha sido posible de desarrollar en un periodo tan corto de tiempo. La elección del framework ha ayudado, pero existen otras alternativas como Cloud Haskell, que permiten programación basada en actores, inspirada en Erlang, que también ayudan.

Las características del lenguaje Haskell, como por ejemplo, su sistema de tipos tan estricto, los *typed holes* y su mantenibilidad, hacen que tanto el desarrollo, como los refactors del proyecto hayan sido fáciles y rápidos.

La facilidad que tiene Haskell de interoperar con otros lenguajes como JavaScript o C (en el caso que hubiera sido necesario) ha ayudado muchísimo a la hora de ayudarnos de librerías externas.

En este proyecto no se han utilizado patrones avanzados como *Free Monads[15]* o *Extensible Effect Systems[16]* para fomentar las futuras contribuciones Open Source.

La programación funcional está tomando un valor cada vez más importante en el día a día, ya que el nivel de concurrencia y distribución se está disparando, y la programación funcional está diseñada para lidiar con esto.

Haskell es el lenguaje de programación funcional por excelencia, y a pesar de que están apareciendo lenguajes como Idris, es uno de los que posee un entorno maduro y adaptado a la programación funcional

11.2.1 Programación concurrente-distribuida con Transient

El grueso de este proyecto es comprender y utilizar el framework Transient para facilitar la programación concurrente y distribuida.

Transient está basado en mónadas especializadas, que permiten añadir comportamientos específicos a las sentencias de código escritas por el usuario.

La definición de una mónada, según Saunders Mac Lane, en el libro *Categories for the Working Mathematician*[17] es (traducida):

Una mónada en [la categoría] X es un monoide en la categoria de endofunctores de X, con producto • reemplazado por composición de endofunctores y el conjunto unidad por el endofunctor unidad.

En el mundo de la programación funcional, la categoría fundamental es la

categoría de los **tipos**, la cual está presente en todos los lenguajes de programación (incluso los dinámicos).

Un **monoide** es un conjunto S , con dos operaciones:

- Operación producto: $\bullet : S \times S \to S$
- Elemento neutro: $e : 1 \to S$

Estas operaciones han de satisfacer las leyes:

- $\forall a, b, c \in S \Rightarrow (a \bullet b) \bullet c = a \bullet (b \bullet c)$
- $\forall a \in S \Rightarrow e \bullet a = a \bullet e = a$

Por otro lado, un **functor**, es un mapeo entre categorías, asociando objetos de una categoría A con otros de una categoría B, tal que:

- $\forall X \in A \Rightarrow \exists F(X) \in B$

Un **endofunctor** es aquel functor que mapea objetos de una categoría con ella misma, lo cual es muy útil en programación, ya que trabajamos sobre la categoría de tipos.

Entonces una **mónada** es

- un endofunctor $T : X- > X$, representado en Haskell como un constructor de tipo $T : * \to *$ con una instancia del tipo de clase **Functor**, que
- posee una transformación natural $\mu : T \times T \to T$ donde $\times$ es la función `join` en Haskell, y además
- posee una transformación natural $\eta : I \to T$, donde I es el endofunctor identidad en la categoría X (o el tipo de tipo $*$ en Haskell). Esta operación en Haskell se conoce como `return`.

Haskell, además introduce un operador $>>=: ma \to (a \to mb) \to mb$ que permite encadenar acciones monádicas. Este operador, junto con `return`, es crucial para que el lenguaje permita ofrecer programación imperativa, ya que es un lenguaje de programación funcional puro.

La programación imperativa en Haskell se basa en la creación de una estructura de datos basada en el concepto de mónadas, donde las acciones se concatenan utilizando un monoide y posteriormente, son evaluadas por el *runtime*.

Por lo tanto, un código imperativo, basado en sentencias, como:

```
main :: IO ()
main = do
    print "Leyendo documento"
    contents <- readFile "documento.txt"
    print contents
```

Pasa por una fase de *desugaring* en el compilador, y pasa a convertirse en la expresión:

```
main :: IO ()
main = print "Leyendo documento"
       >>= \ _ -> readFile "documento.txt"
       >>= \ contents -> print contents
```

Que sigue siendo una expresión, y no sale del concepto funcional.

Gracias a este concepto, los desarrolladores de Haskell, como por ejemplo el autor de Transient, pueden definir sus propios *sub-lenguajes*, añadiendo todo el contexto que necesiten.

En el caso de Transient, el framework define un nuevo tipo monádico `TransIO` que añade contexto a la hora de manejar hilos. Un programa sencillo sería:

```
squares :: TransIO Int
squares = do
    x <- choose [1..100]
    return (x * x)

concurrentMapReduce :: TransIO ()
concurrentMapReduce = do
    sqs <- collect 100 squares
    print (sum sqs)
```

Aquí, `squares` devuelve un numero elevado al cuadrado de la lista `[1..100]`, este numero será utilizado posteriormente por `collect`, que lanza 100 hilos para recoger los numeros devueltos por `squares`, en paralelo.

Al final, se imprime la suma de todos ellos.

Sin que esto fuera suficiente, Transient define otro tipo monádico `Cloud`, basado en `TransIO` que permite ejecutar tareas en diferentes nodos de una red de aplicaciones.

Haskell.do hace un fuerte uso de esto, hasta el punto de ejecutar un nodo en el navegador para la interfaz de usuario y otro nativamente en el sistema, para así poder compilar el código del usuario.

Un ejemplo de esto es el código del evento que activa la compilación:

```
update :: Action -> AppState -> Cloud AppState
update (ToolbarAction Toolbar.Compile) appState = do
    JQuery.show ".dimmedBackground"
    newCompilationState <- atRemote $ Compilation.update ...
    JQuery.hide ".dimmedBackground"
    MathJax.typeset "outputDisplay"
    return appState
        { compilationState = newCompilationState
        }
```

La parte importante de este código es que se ejecuta en el navegador, pero en la **línea 3** se ejecuta una llamada al servidor utilizando la primitiva `atRemote`. Lo mismo se puede hacer a la inversa con código que se encuentra en el servidor, utilizando la primitiva `onBrowser`.

11.2.2 Creación de Ulmus, un microframework para programación funcional reactiva

Transient ofrece de por sí una pequeña librería para generar elementos del DOM en nuestra interfaz gráfica. El problema de esto es que la API que ofrece no permite hacer programación funcional reactiva normalmente y tenemos que modificar elementos uno a uno. Lo que se pretendía desde el principio era trabajar basandonos en la arquitectura Elm[18]:

La arquitectura descrita en la Figura 11.1 nos permite separar nuestra aplicación en tres módulos (representados como rectángulos):

- **View**: El módulo de vista, permite representar la vista como una estructura de datos, para que el runtime se encargue de renderizarla, ya sea en HTML o cualquier tipo de interfaz. Si el usuario interacciona con la UI de alguna manera, por ejemplo pulsando un botón, el runtime se encargará de enviar este evento (que también es una estructura de datos) al módulo de **Update**.
- **Subscriptions**: El módulo de suscripciones, permite suscribir nuestra aplicación a eventos externos, como por ejemplo, un *WebSocket*. El runtime detectará todo tipo de eventos en nuestras suscripciones, y al igual que el módulo de vista, enviará todo tipo de eventos al módulo de **Update**.

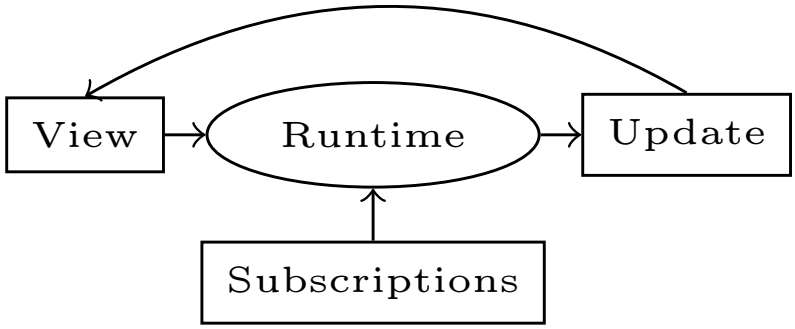

Figura 11.1: Arquitectura Elm

- **Update**: El módulo de actualización de estado. Permite representar la aplicación como una función de los eventos, basándonos en el estado anterior. Al generar un estado nuevo, se envía al módulo de **View**, para que este pueda generar la estructura de vista.

Para poder abordar el problema, se generó un microframework (incluido en el proyecto) llamado Ulmus, que permite estructurar las aplicaciones de esta manera. Realmente se podría resumir en el siguiente código:

```
initializeApp (AppConfig update view updateDisplays initialAppState appPort s
    setup
    webApp appPort $ do
        step view
        loop (step updateDisplays)
  where
    step f = do
        currentState  <- getState initialAppState
        nextAction    <- render $ f currentState
        currentState' <- getState initialAppState
        newState      <- update nextAction currentState'
        local $ setState newState
    loop f = do
        f
        loop f
```

A esta función se le pasan los siguientes parametros:

- `update`. Función que actualiza el estado
- `initialAppState`. Estado inicial de la aplicación
- `appPort`. Puerto por el que se comunicarán la interfaz web y el servidor
- `setup`. Función a ejecutar antes de la interfaz, útil para inicializar algunos parámetros de la aplicación.

Finalmente, los dos parámetros que no se han descrito son los relacionados con la vista.

A diferencia de Elm, en Ulmus se ha utilizado un *Virtual DOM*[19], sino que se ha decido hacer actualizaciones manuales de los elementos. Aprovechando la librería provista por Transient.

Para poder automatizar al máximo las actualizaciones manuales, se ha dejado en la API de Ulmus la posibilidad de pasar una función `view` y otra `updateDisplays`:

- `view` es la función responsable de generar una estructura de datos que representa la vista. Para esto, se ha utilizado la librería `ghcjs-perch`[20], la cual está hecha exactamente para esto.
- Sin embargo, `updateDisplays` es un poco más complicado que esto, ya que se encarga de **actualizar** la vista existente. Por lo tanto el algoritmo de Ulmus **sobrepone** la vista generada por `updateDisplays` a los huecos, además de permitir ejecutar código `IO` en medio:

  ```
  updateDisplays :: State -> Widget ()
  updateDisplays state = do
      Ulmus.newWidget "outputDisplay" (outputDisplay state)
      Ulmus.newWidget "errorDisplay" (errorDisplay state)
      activateScriptTags "#output-frame"
      setHeightFromElement ".error-placeholder" "#errorDisplay"
      highlightCode
  ```

  En este código `Ulmus.newWidget` coloca en el `div` con `id` `outputDisplay` y `errorDisplay` los *displays* pertinentes, que son vistas devueltas por `outputDisplay` y `errorDisplay`.

11.3 Manual del usuario

11.3.1 Requisitos previos

El principal requisito de Haskell.do, es tener instalado `stack`, que es una herramienta de construcción de proyectos Haskell, permite la fácil gestión de dependencias, y a su vez la compilación automática del código fuente de un proyecto.

Para instalar `stack` es suficiente con ejecutar en una terminal UNIX, el siguiente comando:

```
curl -sSL https://get.haskellstack.org/ | sh
```

`stack` es utilizado por Haskell.do como la herramienta de construcción de proyectos, permitiendo así, subirlos al repositorio de paquetes de Haskell, si fuera necesario.

11.3.2 Guía de instalación

1. Instalación binaria Para instalar Haskell.do, podemos obtener una copia binaria desde `http://haskell.do`. Se han de colocar los contenidos del fichero `zip` en un directorio que pertenezca al `PATH` del sistema operativo.
2. Instalación desde código fuente Otra opción es clonar el código fuente desde su repositorio: ** Clonamos con Git el repositorio de GitHub con

   ```
   git clone https://github.com/theam/haskell-do
   ```

   *** Entramos a la raíz del proyecto con** `cd haskell-do` ***** Instalamos GHCJS ejecutando

   ```
   stack setup --stack-yaml=client-stack.yaml
   ```

   ** Construimos el proyecto

   ```
   stack Build.hs -a
   ```

11.3.3 Guía de utilización

A continuación se explica como utilizar la aplicación.

1. **Pantalla principal:**

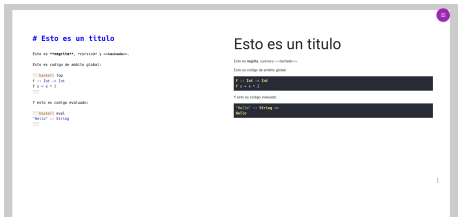

Figura 11.2: Pantalla principal

Según lo que se ve en la Figura 11.2, a la izquierda se encuentra un editor de texto, a la derecha, se encuentra el panel de visualización. Puede escribir código Markdown a la izquierda, encapsulando declaraciones de ámbito global con bloques de código `haskell top` y evaluar expresiones con bloques de código `haskell eval`.

Para evaluar su documento, presione `CTRL + Enter`.

2. **Menú de aplicación:**

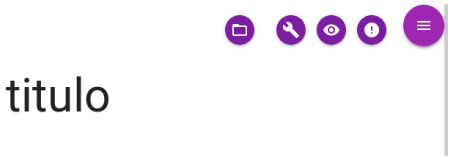

Figura 11.3: Menú de aplicación

El menú de aplicación, según se observa en la Figura 11.3 se despliega situando el cursor del ratón encima. Se ofrecen los botones:

- Crear/Abrir proyecto
- Configurar proyecto
- Ocultar/Mostrar editor de texto
- Ocultar/Mostrar mensaje de error

3. **Abrir/Crear proyecto:**

El diálogo de la Figura 11.4 le permite abrir o crear un proyecto. Para crear un proyecto cree un directorio vacío con el nombre que desea para su proyecto, entre a él y presione el botón de crear proyecto en la esquina inferior derecha.

Figura 11.4: Diálogo de apertura o creación de proyecto

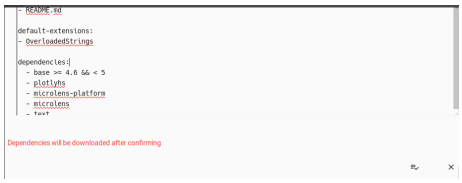

Figura 11.5: Diálogo de configuración de proyecto

4. **Configuración de proyecto y dependencias:**

   La configuración del proyecto se adhiere al estándar `hpack` de Haskell, tras cambiar la configuración del proyecto en el diálogo mostrado por Figura 11.5, Haskell.do comprobará si han cambiado las dependencias, y en este caso, las descargará e instalará.

5. **Creación de gráficas y widgets interactivos:**

Figura 11.6: Gráfica de plotly.js

Si su código devuelve una `String` que contenga HTML, como se ve en la Figura 11.6, Haskell.do automáticamente renderizará este HTML como un widget embebido en el documento. En este ejemplo estamos generando una gráfica de `plotly.js`, la cual incluimos en el HTML del documento utilizando un bloque de código `html_header`.

Capítulo 12

Bibliografía

1. KDnuggets. Analytics, Data Science, Machine Learning Software Poll, top tools share, 2015-2017. https://www.kdnuggets.com/2017/05/poll-analytics-data-science-machine-learning-software-leaders.html
2. Duncan Coutts et al. Stream Fusion. From Lists to Streams to Nothing at All. http://citeseer.ist.psu.edu/viewdoc/download?doi=10.1.1.104.7401&rep=rep1&type=pdf
3. Project Jupyter. http://jupyter.org/
4. Spyder IDE. https://github.com/spyder-ide
5. R Studio. https://www.rstudio.com/
6. Markdown Pad. http://markdownpad.com/
7. IHaskell notebook. https://github.com/gibiansky/IHaskell
8. Hyper Haskell. https://github.com/HeinrichApfelmus/hyper-haskell
9. Zurihac 2017. https://2017.zurihac.info/
10. LambdaWorld 2016. https://www.47deg.com/events/lambda-world-2016/
11. HaskellMAD 2017. https://www.meetup.com/es-ES/Haskell-MAD/events/237823938/
12. Hackage. Haskell documentation. https://hackage.haskell.org/
13. Transient. Haskell framework. https://github.com/transient-haskell/transient

14. QuickCheck. Property based-testing. https://wiki.haskell.org/Introduction_to_QuickCheck1
15. Alexey Avramenko. Free monads explained. https://medium.com/@olxc/free-monads-explained-pt-1-a5c45fbdac30
16. Oleg Kiselyov. Extensible Effects. https://www.cs.indiana.edu/~sabry/papers/exteff.pdf
17. Saunders MacLane. Categories for the Working Mathematician. Editorial Springer
18. Evan Czaplicki. The Elm Architecture. https://guide.elm-lang.org/architecture/
19. Matt Esch. Virtual DOM. https://github.com/Matt-Esch/virtual-dom
20. Arthur S. Fayzrakhmanov. GHCJS-Perch. https://hackage.haskell.org/package/ghcjs-perch