Graduado en Ingeniería Informática

Universidad Politécnica de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

TRABAJO FIN DE GRADO

Entorno de ejecución para un sistema de hojas de cálculo basado en programación funcional.

Autor: Luis Eduardo Bueso de Barrio

Director: Julio Mariño Carballo

MADRID, JULIO DE 2019

Este trabajo está dedicado

A mi hermano, Alejandro.

A mis padres, Marisol y Rafael.

A mi director, Julio.

A mis compañeros y amigos del capítulo de estudiantes de ACM UPM.

Índice general

Resumen VII

Abstract IX

1. Introducción 1
1.1. Historia de las hojas de cálculo . . . 1
1.2. Motivación . . . 2
1.3. Un ejemplo de hoja de cálculo . . . 3
1.4. Propuesta . . . 5

2. Análisis 7
2.1. Conceptos generales de una hoja de cálculo . . . 7
2.2. Ejemplo de uso . . . 8
2.2.1. Análisis . . . 9
2.2.2. Problemas . . . 10

3. Diseño 11
3.1. Objetivos y funcionalidades principales . . . 11
3.2. Arquitectura global del programa . . . 12
3.2.1. Protocolo de comunicación . . . 12
3.3. Diseño del lenguaje de una celda . . . 14
3.4. Diseño de la interfaz gráfica de usuario . . . 14
3.4.1. Comportamiento . . . 15
3.4.2. Componentes y funcionalidades . . . 15
3.5. Diseño del entorno de ejecución . . . 16
3.5.1. Análisis y resolución de dependencias . . . 18
3.5.2. Traducción del contenido de una celda . . . 19
3.5.3. Evaluación del contenido de una celda . . . 20

4. Desarrollo 21
4.1. Trabajos previos . . . 21
4.1.1. Ejemplo de implementación de un componente web sencillo . . . 22
4.1.2. Ejemplo de conexión cliente-servidor utilizando *WebSockets* . . . 25

4.2. Interfaz gráfica de usuario 32
4.2.1. Requisitos previos y compilación 32
4.2.2. Estructura del paquete 33
4.2.3. Estado del componente web 34
4.2.4. Componentes y eventos 35
4.3. Entorno de ejecución 37
4.3.1. Requisitos previos y compilación 37
4.3.2. Estructura del paquete 38
4.3.3. Estado interno 39
4.3.4. Operaciones del servidor 40

5. Resultados **43**
5.1. Ejemplo 43
5.2. Ejemplo con constructores de tipos 45
5.3. Validación de datos 47

6. Conclusiones **51**
6.1. Trabajos futuros 52

Bibliografía **53**

A. Código fuente **55**
A.1. Entorno de ejecución 55
A.1.1. Paquete `Data` 55
A.1.2. Paquete `Lib` 59
A.1.3. Paquete `app` 74
A.2. Interfaz gráfica de usuario 75
A.2.1. Paquete `Component` 75
A.2.2. Paquete `Data.purs` 81
A.3. Programa principal `Main.purs` 85
A.4. Ficheros de marcado y estilo 86
A.4.1. Fichero `app.css` 86
A.4.2. Fichero `index.html` 89

B. Gramática **91**

Índice de figuras

3.1. Arquitectura global del programa 12
3.2. Ejemplo de comunicación. 13
3.3. Evolución del estado interno del cliente 15
3.4. Diseño de la interfaz gráfica de usuario 16
3.5. Diagrama de flujo del servidor 17
3.6. Representación de las dependencias en una hoja de cálculo 18
3.7. Proceso de traducción del contenido de una celda 19

4.1. Componente web del contador 26
4.2. Componente web del contador utilizando *WebSockets* 32
4.3. Interfaz gráfica de usuario 36

5.1. Hoja de cálculo con validadores 48

Índice de cuadros

1.1. Ejemplo de una hoja de cálculo 4

2.1. Ejemplo de hoja de cálculo 8

3.1. Sintaxis de rangos del lenguaje 14

5.1. Ejemplo de hoja de cálculo en el prototipo implementado 43

Resumen

Las hojas de cálculo son una de las herramientas de ofimática más extendidas a nivel de usuario. La importancia de estas herramientas no solo radica en su amplio público, también porque es utilizada como herramienta de toma de decisiones en los consejos de administración de las empresas. Pero a pesar de esta importancia, numerosos estudios han encontrado muchos problemas, que hacen que prácticamente la totalidad de las hojas de cálculo generadas contengan algún tipo de error poniendo en riesgo las decisiones tomadas sobre los datos que contienen. Debido a estos riesgos se han implementado distintas herramientas de análisis para detectar estos errores de la forma más sencilla posible.

En este trabajo se plantea un nuevo enfoque, llevar las ventajas de los sistemas de tipos de los lenguajes de programación funcionales con comprobaciones de tipos estáticas a las hojas de cálculo. La herramienta de hojas de cálculo implementada en este trabajo utiliza un subconjunto de *Haskell* como lenguaje de expresiones de las celdas. Gracias a esto se puede utilizar el sistema de tipos de *Haskell* para mejorar la detección de errores en las hojas. También permite generar tipos de datos compuestos de los datos almacenados en las celdas. Además se ha añadido la funcionalidad de extender las funciones de la hoja de cálculo mediante módulos externos. El resultado obtenido es un prototipo completamente funcional capaz de realizar todas las tareas descritas.

Palabras clave: programación funcional, *Haskell*, hoja de cálculo, entorno de ejecución.

Abstract

Spreadsheet programs are one of the most widespread information processing tools used by end-users. The importance of these tools is not only in their extensive use but because of its relevance as a decision-making tool by executives. Besides this, some studies have found plenty of problems that make spreadsheets widely faulty, which make very risky making decisions on the data they contain. Because of these risks some analysis tools have been implemented to find errors easily.

This project presents a new approach, to take the advantages from type systems used in functional programming languages with static typing to spreadsheets. The implemented tool will use a subset of *Haskell* as the language for its cell expressions. This unlocks *Haskell*'s type system to improve error detection. It also allows generating data types from the data stored in the table. The tool provides an easy mechanism to extend the set of functions allowed on the spreadsheets. The mechanism to do this is the inclusion of external modules. The final result of this project is a completely functional prototype that can perform any of the described tasks.

Keywords: functional programming, *Haskell*, spreadsheet, runtime.

1 Introducción

Una hoja de cálculo es un tipo de documento que permite la manipulación de datos numéricos y alfanuméricos dispuestos en forma de tablas. Estas tablas están compuestas por celdas que se organizan en una matriz bidimensional.

1.1 Historia de las hojas de cálculo

La historia [10] de las hojas de cálculo está íntimamente relacionada con la historia de la computación personal. Estos productos han permitido acercar la potencia de la computación a usuarios sin conocimientos en programación a través de tablas interactivas que se afrontan más como un "dispositivo de resolución de problemas" que como un sistema de programación.

Una de las características más importantes que tienen todos estos programas y los hace tan intuitivos es el cálculo casi instantáneo de las fórmulas. Están inspirados en la filosofía *WYSIWYG*[1] también ampliamente utilizada por herramientas como los procesadores de texto. Esta filosofía busca visualizar el resultado de la manipulación de los datos del programa de forma instantánea, facilitando su uso por parte de los usuarios.

A pesar de lo que pueda parecer, los programas de hojas de cálculo se han mantenido muy constantes durante toda su historia, normalmente añadiendo funcionalidades discretas y especialmente relacionadas con el avance en sistemas de interfaces gráficas de usuario.

El primer programa de hojas de cálculo fue publicado en 1979 [10]. Este programa recibió el nombre de *VisiCalc* y apostó claramente por la filosofía de diseño *WYSIWYG*.

[1] Acrónimo de *What You See Is What You Get*

1 Introducción

Tras el abrumador éxito de *VisiCalc*, empresas competidoras comenzaron el desarrollo de “clones”. En 1983 se publicó *Lotus 1-2-3* [10] que añadió un sistema de macros al programa que permitía la manipulación de los datos de una forma más rápida. Además apostó por añadir *kits* de desarrollo que permitían crear extensiones personalizadas.

En 1985 *Microsoft* publica *Excel*, que aprovecha la interfaz gráfica de usuario del *Macintosh* para ser más amigable. Tiempo más tarde los ordenadores de *IBM* comenzaron a soportar interfaces de usuario similares a las ofrecidas en el ordenador de *Apple*, lo que permitió llevar estos avances en interfaz gráfica de usuario a estos ordenadores.

Gracias al éxito de *Windows* y el ecosistema de aplicaciones de *Microsoft*, el mercado ha sido completamente dominado por *Excel* hasta hoy.

En 2006, *Google* presentó su programa de hojas de cálculo, *Google SpreadSheets*. Este programa de hojas de cálculo se pensó como un “complemento” al programa de *Microsoft*, que contaba solamente con un subconjunto de las funcionalidades de *Excel*.

La historia de las hojas de cálculo se ha centrado en la búsqueda de un producto estandarizado. Su éxito ha sido provocado por una filosofía de diseño que ha facilitado enormemente su uso.

1.2 Motivación

Las hojas de cálculo se han convertido en uno de los programas más utilizados a nivel global. El programa de hojas de cálculo *Microsoft* contaba con 750 millones de usuarios en 2010 [8].

Además las hojas de cálculo son una forma muy popular de toma de decisiones en los consejos de administración en empresas. Algunos datos representativos [2] de su uso son:

- La firma de inteligencia financiera *CODA* reportó que el 95 % de las empresas estadounidenses utilizan hojas de cálculo para reportar datos financieros.
- En 2004 la *IDC*[2] entrevistó a 118 empresas líderes en el sector. Los resultados fueron que el 85 % de las empresas estaban utilizando hojas de cálculo para hacer presupuestos y reportar ganancias.
- En más entrevistas a ejecutivos financieros realizadas en 2004 reveló que las tecnologías más utilizadas para realizar presupuestos eran las hojas de cálculo.

Todos estos datos revelan la importancia de las hojas de cálculo como herramienta, no sólo por la cantidad de usuarios que tienen, también como herramienta de toma de decisiones en las empresas.

Debido a la importancia de las hojas de cálculo se han realizado numerosos estudios sobre los errores que aparecen en este tipo de herramienta [2] [3] [4].

[2] *International Data Corporation*

Se estima que el 94 % de las hojas de cálculo contienen algún tipo de error y que el 5,2 % de las celdas de una hoja de cálculo contienen errores [2]. Algunas historias ocasionadas por errores en hojas de cálculo [9]:

- Candidatos a oficial de policía recibieron un aprobado en las pruebas cuando en realidad habían suspendido. La razón fue que los datos de la hoja de cálculo estaban mal ordenados.
- Un colegio pierde 30.000£ por estimar mal su presupuesto. La razón fue que los números fueron introducidos como texto en la hoja de cálculo.
- Los beneficios de una empresa de telecomunicaciones se calculan erróneamente, el resultado es que los datos obtenidos son 50 millones de dólares inferiores a los beneficios reales. La razón fue la existencias de referencias incorrectas en una fórmula.

Para evitar y detectar los errores en las hojas de cálculo se han llevado a cabo gran cantidad de trabajos. Uno de ellos [5] es el diseño de una metodología de desarrollo para mejorar la calidad y el diseño de las hojas de cálculo. Los objetivos principales de esta metodología son obtener hojas de cálculo fiables, con resultados correctos y consistentes, que además permitan trazar errores, y que sean fácilmente modificables sin inducir a nuevos errores. Otra de las aproximaciones que se han llevado a cabo para facilitar la detección de errores en hojas de cálculo es el desarrollo de herramientas de análisis estático [6].

Este tipo de herramientas han mostrado una mejora sustancial en la calidad de las hojas de cálculo en la que se han aplicado. El origen de estas metodologías de desarrollo y herramientas de análisis estático está claramente influenciado por el mundo del *software*.

El problema de la eliminación de errores en el *software* ha existido desde los comienzos de la era informática. El desarrollo de lenguajes de programación ha estado íntimamente ligado con el proceso, evolucionando hacia lenguajes con sistemas de tipos que permiten la detección de errores en los primeros momentos de la compilación de un programa. Estudios recientes [7] han mostrado que los lenguajes funcionales con sistemas de tipos estáticos afectan positivamente a la calidad del *software* desarrollado.

La motivación de este trabajo es mejorar los sistemas de hojas de cálculo actuales llevando las ventajas de los lenguajes de programación funcionales con comprobaciones de tipos estáticas a estas. Con esto se pretende mejorar la calidad y la detección de errores en las mismas.

1.3 Un ejemplo de hoja de cálculo

La principal aplicación de las hojas de cálculo es el tratamiento y análisis de datos. Sus usos se extienden desde el cálculo la contabilidad en una empresa hasta el cálculo de la media de una asignatura. En este ejemplo se muestra una hoja de cálculo que reúne dife-

rentes datos sobre las asignaturas que un estudiante ha ido obteniendo durante la carrera[3].

	A	B	C	D	E	G	H
1	**Curso**	**Asignatura**	**Créditos**	**Nota**	**Calificación**	**Total**	60
2	1	Álgebra lineal	6	5	Aprobado	Créditos 1	30
3	1	Calculo	6	7,1	Notable	Créditos 2	6
4	1	Lógica	6	7	Notable	Créditos 3	9
5	1	Programación I	6	9	Sobresaliente	Créditos 4	15
6	1	Programación II	6	8,8	Notable		
7	2	Concurrencia	3	7,8	Notable	**Media**	7,95
8	2	Bases de Datos	3	5,4	Aprobado	Media 1	7,38
9	3	*Middleware*	3	8	Notable	Media 2	6,6
10	3	Sistemas Operativos	6	8,1	Notable	Media 3	8,05
11	4	*Practicum*	12	9,6	Sobresaliente	Media 4	9,56
12	4	Computabilidad	3	9,4	Sobresaliente		

Cuadro 1.1: Ejemplo de una hoja de cálculo

En este ejemplo los datos de las asignaturas se han clasificado en columnas. Los datos almacenados incluyen las columnas de la A a la E. Los datos almacenados se corresponden a:

- El curso al que pertenece la asignatura, almacenado en la columna A. Se registra como un valor numérico. Sus valores deberían oscilar entre 1 y 4.
- El nombre de la asignatura, almacenado en la columna B. Corresponde a una cadena de caracteres alfanuméricos.
- El número de créditos que contiene cada asignatura, almacenado en la columna C. También corresponde a un valor numérico. Sus valores nunca deberían ser números negativos.
- La nota obtenida sobre esa asignatura, almacenada en la columna D. Se registra con un número entero. Los valores de estos datos no deberían ser inferiores a 0 ni superiores a 10.
- La calificación de la asignatura, almacenada en la columna E. Se representa con una cadena de caracteres. Los valores que deberían contener únicamente las cadenas "Suspenso", "Aprobado", "Notable" o "Sobresaliente" y deberían calcularse en función a la nota almacenada en la columna anterior.

Sobre estos datos se han realizado dos pequeños análisis en las columnas G y H. En primer lugar se ha analizado el número de créditos totales y por curso de los datos reflejados en la tabla. El segundo análisis realizado se corresponde a las medias ponderadas por número de créditos.

[3]Para simplificar el ejemplo se ha realizado sobre 12 asignaturas

Para expresar estos análisis las herramientas de hojas de cálculo son capaces de evaluar expresiones contenidas en las celdas. Estas expresiones permiten operar con el contenido de otras celdas a través de referencias. Además en una fórmula no se permite la modificación de celdas ajenas ni de ocasionar otros efectos. Esta es una de las características principales de los lenguajes funcionales.

1.4 Propuesta

El objetivo principal de este trabajo es el diseño y la implementación de un entorno de ejecución para un sistema de hojas de cálculo funcionales. El prototipo resultante deberá ser capaz de evaluar correctamente código de un lenguaje funcional con sistema de tipos estáticos en cada una de las celdas del lenguaje. El lenguaje que se utilizará como lenguaje de expresiones será un subconjunto del lenguaje de programación *Haskell*.

La interfaz gráfica de usuario de este entorno de ejecución deberá contar con los elementos básicos de una herramienta de hojas de cálculo actual. Los elementos básicos serán tres: celdas en el interior de una tabla; una fórmula general que muestre la entrada completa introducida en una celda; un cuadro de texto que muestre el último resultado de la celda seleccionada.

Como requisito de usabilidad, la interfaz gráfica del prototipo deberá mantener la filosofía *WYSIWYG*, ya que la popularidad y facilidad de uso de las hojas de cálculo ha venido dada en gran parte por esta filosofía de funcionamiento. Para conseguir este efecto la interfaz actualizará instantáneamente el resultado de una celda una vez se haya finalizado la introducción de texto.

Además, se ha propuesto añadir la funcionalidad de extender las funciones que se puedan utilizar en la hoja de cálculo a través de módulos externos. De esta forma podrán extenderse las funcionalidades de las fórmulas de un modo sencillo. La interfaz gráfica contará con un panel de texto que se utilizará para introducir código fuente que podrá utilizarse en las fórmulas de las celdas, extendiendo las funciones disponibles por defecto.

El uso de *Haskell* como lenguaje de las fórmulas permitirá utilizar las ventajas que ofrece su sistema de tipos para la detección prematura de errores que ahora son muy difíciles de detectar en las herramientas actuales.

Mediante el uso de constructores de tipos se pueden diferenciar los datos numéricos presentes en el ejemplo 1.1. Haciendo esta diferenciación se consigue que el sistema de tipos detecte errores sobre los datos utilizados en las fórmulas. Por ejemplo la suma de un crédito y un curso generaría un error, ya que pertenecen a tipos diferentes (un dato sería tipo “Crédito” y otro dato sería tipo “Curso”). Para operar correctamente estos tipos debe hacerse una conversión explícita.

Pero la potencia del sistema de tipos de *Haskell* no termina aquí, también permitirá generar tipos compuestos utilizando los datos de las celdas y realizar validaciones de los datos al generar estos tipos compuestos. Por ejemplo, en los datos anteriores podría validarse

un tipo compuesto “Asignatura” si cada uno de los datos que forman el tipo compuesto (curso, número de créditos, nota y calificación) cumplen ciertas propiedades (nota entre 0 y 10, curso entre 1 y 4...).

Estructura del documento

En el capítulo 2 se realiza un análisis del funcionamiento de las hojas de cálculo a través de un ejemplo de uso y se analizan los problemas que aparecen al implementar una herramienta de hojas de cálculo. En el capítulo 3 se detalla el diseño global de la herramienta y se plantean soluciones de diseño para los problemas encontrados en la fase de análisis. En el capítulo 4 se detalla el proceso de desarrollo del proyecto. En el capítulo 5 se analiza el prototipo generado tras el desarrollo comparándolo con las tecnologías de hojas de cálculo existentes. Finalmente en el capítulo 6 se analiza el prototipo obtenido y se plantean líneas futuras de desarrollo.

2

Análisis

En este capítulo se definen los conceptos generales necesarios para tratar con una hoja de cálculo. A continuación se desarrolla un análisis de las funcionalidades que ofrecen las hojas de cálculo a través de un ejemplo y finalmente se enumeran distintos problemas que habrá que resolver durante el diseño y el desarrollo del programa.

2.1 Conceptos generales de una hoja de cálculo

Las hojas de cálculo pueden definirse como documentos que contienen grandes matrices cuyas columnas están identificadas por letras y sus filas por números [5]. Los conceptos esenciales para comprender el funcionamiento de una hoja de cálculo son:

- **Celda**: es la unidad básica de entrada de datos en una hoja de cálculo. Las celdas se identifican por la combinación de una letra que identifica su columna y un número que identifica su fila. Las celdas pueden contener en su interior valores numéricos, valores alfanuméricos, referencias a otras celdas o combinaciones de todo lo anterior a través de fórmulas. El concepto más importante es que cualquier expresión introducida en una celda se evalúa a un único "valor".

- **Fórmula**: es la expresión contenida en una celda. Las fórmulas pueden contener llamadas a funciones, referencias al contenido de otras celdas, valores constantes... Las fórmulas obedecen una gramática determinada por cada programa de hoja de cálculo, que define las expresiones válidas de una celda. Una vez la expresión de la celda es evaluada, muestra en la celda que la contiene un resultado o un error.

- **Referencia**: es una etiqueta que puede sustituirse por el resultado de la celda a la que apunta. Las referencias identifican una celda concreta a través de su "índice" en

la matriz, formado por la letra que identifica la columna y el número que identifica la fila. En los sistemas de hojas de cálculo actuales se distingue entre dos tipos de referencias:

- Referencias absolutas: que realmente apuntan siempre a la celda que describen.
- Referencias relativas: que identifican un desplazamiento con respecto a la celda en la que se encuentran.

Además, las hojas hojas de cálculo modernas cuentan con numerosas funciones auxiliares [1] que aumentan las funcionalidades de las hojas de cálculo y sus aplicaciones. Algunas de estas funcionalidades son tratamiento de fechas, funciones de ingeniería, funciones lógicas...

2.2 Ejemplo de uso

	A	B	C	D	E	G	H
1	**Curso**	**Asignatura**	**Créditos**	**Nota**	**Calificación**	**Total**	60
2	1	Álgebra lineal	6	5	Aprobado	Créditos 1	30
3	1	Calculo	6	7,1	Notable	Créditos 2	6
4	1	Lógica	6	7	Notable	Créditos 3	9
5	1	Programación I	6	9	Sobresaliente	Créditos 4	15
6	1	Programación II	6	8,8	Notable		
7	2	Concurrencia	3	7,8	Notable	**Media**	7,95
8	2	Bases de Datos	3	5,4	Aprobado	Media 1	7,38
9	3	*Middleware*	3	8	Notable	Media 2	6,6
10	3	Sistemas Operativos	6	8,1	Notable	Media 3	8,05
11	4	*Practicum*	12	9,6	Sobresaliente	Media 4	9,56
12	4	Computabilidad	3	9,4	Sobresaliente		

Cuadro 2.1: Ejemplo de hoja de cálculo

Los datos en esta tabla están ordenados verticalmente en columnas. La fila 1 se ha utilizado para etiquetar los datos de las columnas inferiores. De la columna A a la columna D hay datos simples que indican el curso, el nombre de la asignatura, el número de créditos que tiene y la nota que se ha obtenido.

En la columna E están los datos de la calificación. Para evitar errores estos datos se generan en función de la nota obtenida a través del condicional:

```
E2 = IF ( D2<5 ; "Suspenso" ;
       IF ( D2<7 ;  "Aprobado" ;
         IF ( D2<9 ; "Notable" ;
           IF ( D2<=10 ; "Sobresaliente" ; "Suspenso" ))))
```

Esta fórmula contiene cuatro condicionales anidados sobre el contenido de la celda $D2$. Este condicional no es capaz de detectar errores en el dato "notas", ya que si tenemos una nota que se sale de los rangos habituales de las calificaciones[1] este condicional lo pondría como "Suspenso". Esto se puede arreglar acotando los valores de las calificaciones utilizando la función AND y devolviendo "Error" en caso de que no se encuentre en ningún tramo de la calificación.

```
E2 = IF ( AND ( D2<5 ; D2>=0 ) ; "Suspenso" ;
       IF ( AND ( D2<7; D2>=5 ) ; "Aprobado" ;
         IF ( AND ( D2<9 ; D2>=7 ) ; "Notable" ;
           IF ( AND ( D2<=10 ; D2>= 9 ) ; "Sobresaliente" ; "Error" ))))
```

En cada una de las celdas inferiores habrá que modificar este condicional para las celdas que le correspondan (desde la $D2$ hasta la $D12$ en este ejemplo). Esto también podría generar problemas, por ejemplo si no se renombra correctamente una de las 4 etiquetas que tiene esta fórmula para cada una de las celdas en las que es necesaria.

En la celda $H1$ se almacena el sumatorio de la columna de créditos mediante la fórmula:

```
H1 = SUM ( C:C )
```

Esta fórmula es bastante sencilla, hace uso de la función SUM que realizará el sumatorio de la columna dada. En las celdas $H2$, $H3$, $H4$ y $H5$ se utiliza una versión especializada de esta fórmula que realiza previamente un filtrado por el curso:

```
H2 = SUM ( ( FILTER ( C:C ; A:A=1 ) ) )
```

Finalmente, las fórmulas más complejas de esta tabla son las que realizan las medias ponderadas por crédito de las asignaturas. La media global de todas las asignaturas se realiza mediante:

```
H7 = SUMPRODUCT ( C:C ; D:D ) / SUM ( C:C )
```

Haciendo uso de la función $SUMPRODUCT$, que realiza primero los productos de los datos de las columnas C y D y posteriormente la suma del resultado.

Como en el caso de los créditos, finalmente se realiza una versión especializada de la fórmula anterior a través de:

```
H8 = SUMPRODUCT ( FILTER ( C:C ; A:A=1 ) ; FILTER ( D:D ; A:A=1 ) )
       / SUM ( ( FILTER ( C:C ; A:A=1 ) ) )
```

Obteniendo así la media ponderada por curso.

2.2.1 Análisis

A pesar de que este ejemplo pueda parecer bastante consistente a simple vista puede generar bastantes errores. En primer lugar todas las referencias utilizadas son relativas.

[1] del 1 al 10

Esto implica que si en algún momento se copia alguna fórmula en otras celdas dejará de funcionar como se espera.

Para solventar este problema pueden escribirse todas las etiquetas con el símbolo $ así se convierten en referencias absolutas y se asegura su consistencia en caso de copiarse a otras celdas.

Las hojas de cálculo tradicionales tienen sistemas de validación de datos [1] mediante predicados sencillos. Con este tipo de predicados se puede verificar por ejemplo que todos los datos introducidos en la columna de notas sean del 1 al 10, solventando el problema que se ha encontrado en el condicional que genera la calificación.

A pesar de esto, esta validación no nos puede asegurar que cuando realizamos un filtro, no mezclemos “notas” y “créditos”, ya que en ambos casos los valores son similares. Este error podría detectarse utilizando los sistemas de tipos que ofrecen los lenguajes funcionales estáticos como *Haskell*.

2.2.2 Problemas

Durante el desarrollo de este ejemplo también se han encontrado distintos problemas que habrá que resolver para tener un entorno de ejecución que funcione correctamente.

Uno de ellos es la detección de referencias circulares, que deberán ser detectadas para no generar problemas de evaluación infinita de expresiones. Estas evaluaciones infinitas también pueden aparecer en las expresiones de las celdas si se utiliza un lenguaje *Turing-completo*.

El lenguaje de expresiones de las celdas deberá extenderse para soportar referencias entre celdas y rangos, igual que las hojas de cálculo tradicionales.

El manejo de celdas vacías será un punto clave del diseño del entorno de ejecución. Actualmente cada herramienta de hojas de cálculo tiene distintos mecanismos para lidiar con celdas vacías [1], muchos de ellos parecen ser bastante problemáticos tratándolas con valores por defecto y provocando que ciertos errores sean muy difíciles de trazar.

También deberán detectarse errores de tipos en las celdas cuando las funciones reciban tipos incorrectos.

En cuanto a la interfaz gráfica de usuario de este entorno de ejecución, deberá ser bastante rápida y en ningún momento bloquearse a la espera de resultados, para conseguir el diseño *WYSIWYG* que ha hecho tan populares a este tipo de programas.

Durante el análisis se ha decidido que el diseño del prototipo se centrará en las funcionalidades esenciales de las hojas de cálculo olvidando complementos como formato de celda, acciones de “arrastrar” celdas... etc. En la sección 3.4 se concretarán los componentes que tendrá la interfaz gráfica de usuario.

3

Diseño

En este capítulo se tratan las decisiones de diseño tomadas durante el desarrollo del proyecto. Además, se explica la arquitectura del programa, el diseño del entorno de ejecución, el lenguaje soportado por las celdas, el análisis de referencias, el proceso de evaluación de las celdas y el diseño de la interfaz gráfica de usuario.

3.1 Objetivos y funcionalidades principales

El objetivo principal del proyecto es implementar un prototipo funcional del entorno de ejecución de unas hojas de cálculo basado en programación funcional.

Las funcionalidades básicas que deberá tener el prototipo son:

- Evaluación correcta del contenido de una celda.
- Detección de errores en el contenido de una celda.
- Resolución de referencias entre celdas, también debe ser capaz de detectar problemas como referencias circulares entre celdas.
- Extensión de las funcionalidades básicas mediante módulos externos.
- Guardado y carga del contenido de la hoja de cálculo en ficheros externos.

3.2 Arquitectura global del programa

Para simplificar la implementación del proyecto, el entorno de ejecución se dividirá en dos programas:

- Una interfaz web que hará la función de interfaz gráfica y con la que interactuará el usuario.
- Un programa servidor que hará el cómputo completo de las hojas de cálculo.

La conexión entre ambas partes se realizará a través de *WebSockets* [12] como se muestra en la figura 3.1.

Figura 3.1: Arquitectura global del programa

De este modo se conseguirá completa independencia entre la interfaz de usuario y el programa que realizará el cómputo, pudiendo actualizar la interfaz gráfica o el programa servidor siempre y cuando se siga el protocolo que se establece en la sección 3.2.1

3.2.1 Protocolo de comunicación

El protocolo de comunicación entre el programa cliente y el programa servidor consta de varias operaciones:

- **Carga de un fichero:** que leerá el fichero de la ruta especificada en el mensaje y actualizará el estado del servidor.
- **Evaluación de una celda:** que enviará el contenido de una celda al servidor. Este calculará el resultado y lo devolverá al cliente.
- **Carga de un módulo externo:** que actualizará el entorno de ejecución completo añadiendo el nuevo módulo.
- **Guardado de un fichero:** que almacenará el estado actual de la hoja de cálculo y los módulos cargados en un fichero localizado en la ruta especificada en el mensaje.

Todas las operaciones de este protocolo serán asíncronas, es decir, en ningún momento se produce un bloqueo del cliente para esperar un resultado del servidor. De esta forma se evita que el usuario tenga que esperar en caso de que una celda tenga una evaluación más larga.

En la figura 3.2 se muestra un ejemplo de comunicación durante una sesión de uso de las hojas de cálculo. En este ejemplo se muestran las siguientes operaciones:

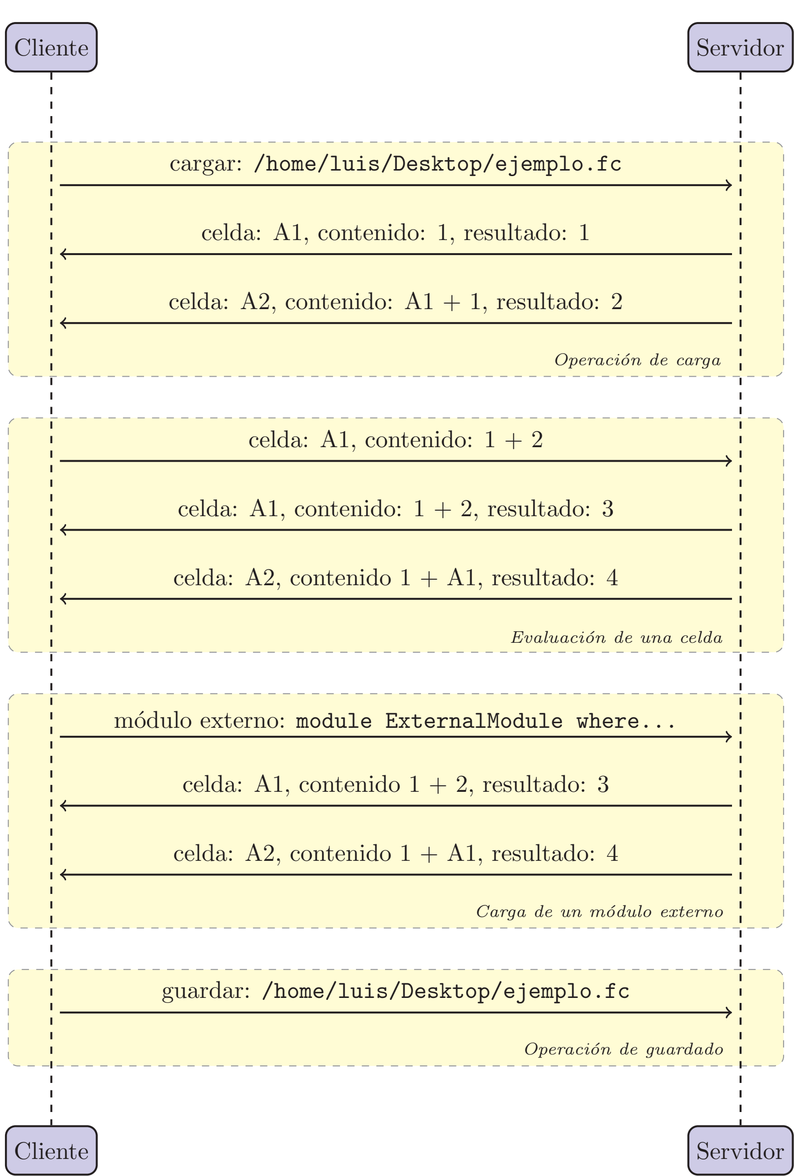


Figura 3.2: Ejemplo de comunicación.

- Operación de carga del fichero `/home/luis/Desktop/ejemplo.fc`: la operación de carga provocará el envío de N mensajes, donde N representa el número de celdas que hay almacenadas en el estado guardado en el fichero `ejemplo.fc`.
- Operación de evaluación de la celda $A1$: el cliente modifica el contenido de la celda $A1$, introduciendo $1 + 2$. Tras ser evaluado el servidor emite un mensaje con el resultado, 3, y a continuación emite N mensajes, donde N representa cada una de las celdas que hacen referencia a la celda modificada, $A1$. En este caso la única celda que hace referencia a $A1$ es la celda $A2$.
- Operación de carga de un módulo externo: cuando el cliente modifica el módulo externo, este se añade al entorno de ejecución. A continuación se producen N mensajes, uno por cada celda con contenido en el estado del servidor.

- Operación de guardado: esta operación no genera mensajes extra entre el servidor y el cliente y almacena el estado actual del servidor en el fichero especificado en la ruta del mensaje.

3.3 Diseño del lenguaje de una celda

El lenguaje soportado por las celdas será un subconjunto de expresiones del lenguaje *Haskell*. La gramática del lenguaje soportado se encuentra en el apéndice A.4.2.

Las referencias soportadas en las expresiones serán las mismas que tienen los lenguajes tradicionales [1]:

Rango a seleccionar	Sintaxis utilizada
Una sola fila	`1:1`
Una sola columna	`A:A`
Celdas contiguas	`A1:C5, F1:F5`
Intersección de rangos	`A1:C5 F1:F5`

Cuadro 3.1: Sintaxis de rangos del lenguaje

La sintaxis de los rangos se extenderá con dos tipos de operaciones en los rangos:

- `[]` que introducirá los valores de las celdas representados por ese rango en una estructura de datos, por defecto esa estructura será una lista. Un ejemplo de uso de esta sintaxis sería `sum [A1:A4]`, que introduciría los valores almacenados en las celdas definidas por el rango `A1:A10` en una lista `[A1, A2, A3, A4]` y se aplicaría a la función `sum`.
- `()` que pasará los valores de las celdas representados por ese rango como parámetros. Con esto se conseguirá utilizar rangos para construir tipos de datos compuestos en otras celdas de forma sencilla.

Dentro de la sintaxis de las celdas se ha decidido dar soporte únicamente a referencias absolutas. Por tanto la sintaxis básica[1] no será de referencias relativas. Se deja abierto a darle soporte en un futuro, pero se utilizará una sintaxis distinta a la “estándar” ya que su interpretación puede llevar a confusiones y generar errores.

3.4 Diseño de la interfaz gráfica de usuario

En esta sección se explicará el diseño del programa cliente que realizará la función de interfaz gráfica de usuario. Se tratarán sus funcionalidades principales, su comportamiento

[1] por ejemplo $A1$ sin $

y los componentes esenciales que contendrá.

3.4.1 Comportamiento

En la figura 3.3 se ilustra el comportamiento del cliente y la evolución de su estado interno al recibir distintas entradas del usuario.

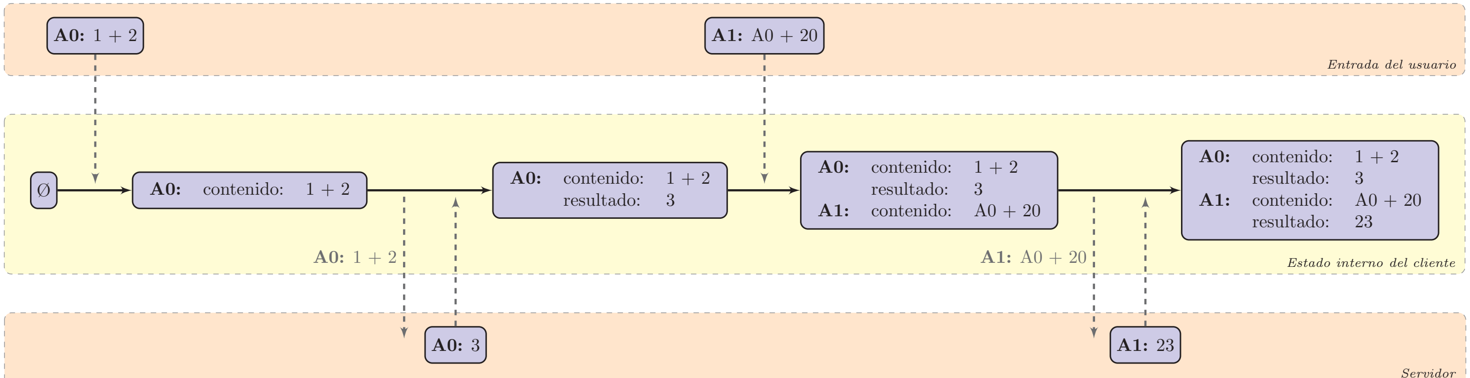


Figura 3.3: Evolución del estado interno del cliente

Al recibir contenido de una celda, lo almacena en su estado interno y lo envía al servidor. Este calcula el resultado y lo envía de nuevo al cliente, que almacenará este resultado en su estado interno.

El comportamiento de la interfaz gráfica de usuario en ningún momento será bloqueante y el refresco del resultado generado por el servidor será completamente instantáneo para conseguir la experiencia *WYSIWYG*.

3.4.2 Componentes y funcionalidades

La interfaz gráfica de usuario deberá permitir al usuario las siguientes acciones:

- Introducir, borrar y modificar el contenido de una celda.
- Mostrar el resultado de la evaluación de una celda.
- Introducir, borrar y modificar el contenido del módulo externo.
- Introducir rutas para cargar y guardar archivos con el contenido de la hoja de cálculo.

Los componentes necesarios para realizar estas acciones son:

- Una tabla con celdas que permitan la entrada de texto.

fx : 7

	A	B	C	D	E
0	**Curso**	**Asignatura**	**Créditos**	**Nota**	
1	1	Álgebra lineal	6	5	
2	1	Cálculo	6	7|	
3	1	Lógica	6		
4	1	Programación I	6		
5	1	Programación II	6		
6	2	Concurrencia	3		
7	2	Bases de Datos	3		
8	3	*Middleware*	3		
9	3	Sistemas Operativos	6		
10					
11					
12					
13					
14					
15					
16					
17					

```
module External module where
```

Result : 7

Figura 3.4: Diseño de la interfaz gráfica de usuario

- Un cuadro de texto donde se muestre el resultado o el error producido tras evaluar una celda.
- Un cuadro de texto que muestre la fórmula introducida en una celda.
- Un área de texto que permita escribir el módulo externo.
- Un área de texto que permita escribir la ruta del fichero que se desee cargar o guardar.
- Botones que permitan realizar la acción de cargar o guardar el fichero.

3.5 Diseño del entorno de ejecución

En esta sección se va a explicar el diseño del servidor, las tareas principales que realizará y su flujo de funcionamiento. Las tareas principales que tendrá el servidor son:

- Proceso de traducción para eliminar el azúcar sintáctico.
- Análisis de dependencias de una celda.
- Resolución de dependencias de una celda.
- Evaluación de una celda.
- Actualización del estado global de la hoja de cálculo.

- Actualización de las celdas dependientes.
- Carga del estado de una hoja de cálculo de un fichero.
- Guardado del estado de la hoja de cálculo en un fichero.

El funcionamiento global del servidor se muestra en la figura 3.5.

En primer lugar el servidor deberá crear un fichero en el que se guardará el contenido del módulo externo introducido por el usuario y abrir el *WebSocket* al que se conectará la interfaz gráfica de usuario. Esta fase será la fase de configuración del servidor.

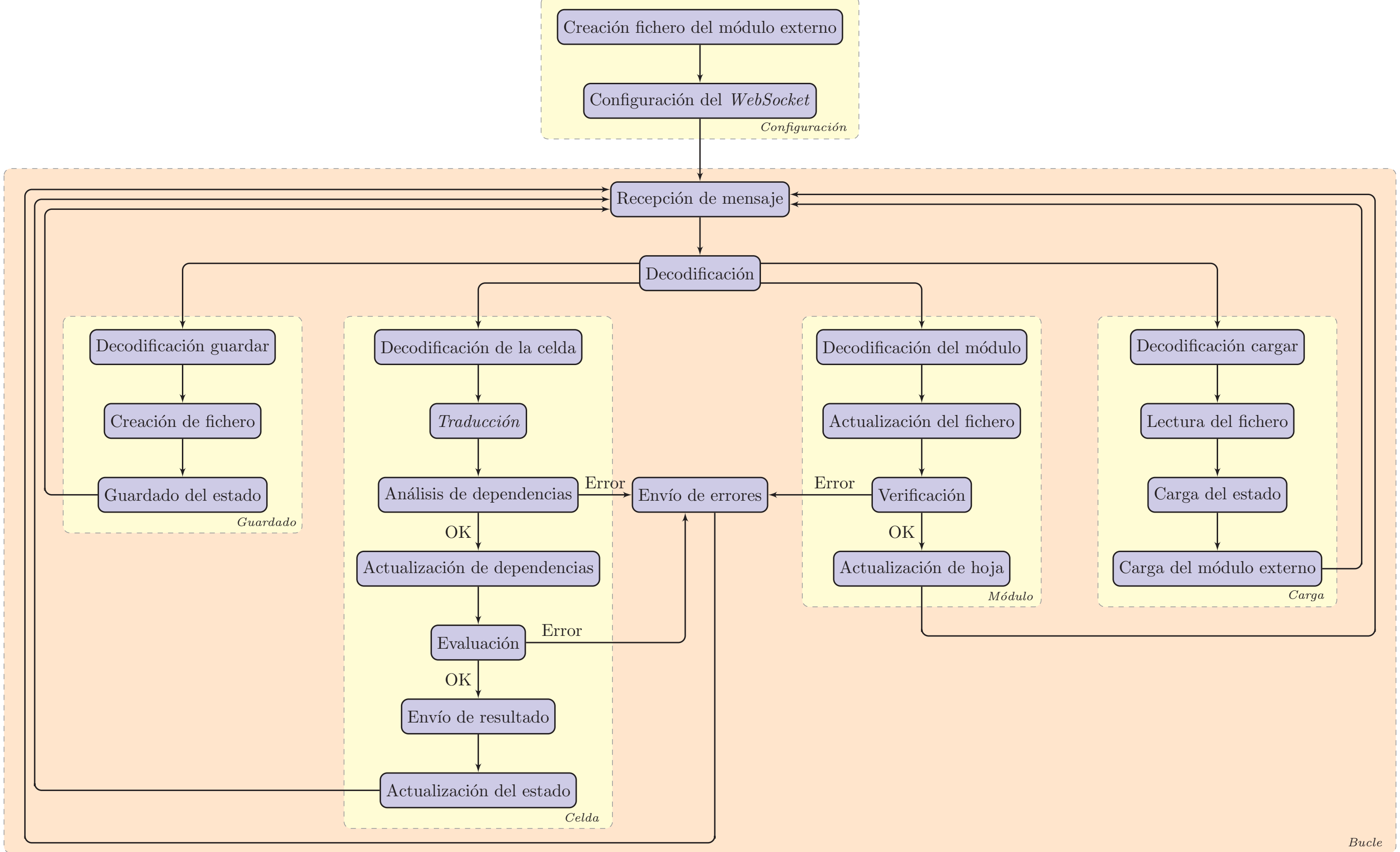


Figura 3.5: Diagrama de flujo del servidor

Tras finalizar la configuración, el servidor realizará continuamente un bucle, que en función de lo recibido por el *WebSocket* realizará las siguientes tareas:

- **Guardado** del estado actual del servidor en la ruta especificada en el mensaje recibido. Para ello creará un fichero en la ruta dada y a continuación escribirá el estado.
- **Carga** del estado actual del servidor en la ruta especificada en el mensaje recibido. Esta operación consiste en la lectura del fichero en la ruta proporcionada por el mensaje. A continuación se cargará el estado almacenado en ese fichero. Finalmente

se restablecerá el módulo externo escrito por el usuario, también almacenado en ese fichero.

- **Evaluación** de una celda. En primer lugar se realizará la traducción descrita en la sección 3.5.2. Una vez se ha realizado esta traducción, se analizarán las referencias buscando referencias circulares, en caso de no encontrarse problemas se almacenarán en el grafo de dependencias descrito en la sección 3.5.1. A continuación se evaluará la expresión obtenida anteriormente. Finalmente se reporta el resultado enviándolo a través del *WebSocket*.

- Carga de un **módulo externo**. Esta operación consistirá en la actualización del fichero que contiene el módulo externo. Después se verificará que el módulo se carga correctamente y si es así se volverá a actualizar el contenido de todas las celdas de la hoja de cálculo.

3.5.1 Análisis y resolución de dependencias

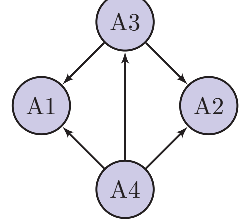
Una parte vital del funcionamiento de las hojas de cálculo son las referencias entre celdas. Para su correcto funcionamiento, se deben evitar referencias circulares que generen ciclos infinitos de evaluación de celdas.

Para almacenar el estado de las dependencias entre celdas se utilizará un grafo dirigido como el mostrado en la figura 3.6. En el contenido de este ejemplo se muestra el resultado de un grafo en el que hay celdas:

- Con un único valor en su interior, como las celdas $A1$ y $A2$. Este contenido generará vértices sueltos en el grafo.

- Con operaciones sobre otras celdas, que se reducen a un único valor, como la celda $A3$. Este contenido generará aristas que unen el vértice que representa esta celda con cada uno de los vértices que representan las celdas de las que depende.

- Con rangos, que se traducirán a una expresión *Haskell* siguiendo el mecanismo explicado en la sección 3.5.2. Una vez realizada la traducción, se tratará igual que el caso anterior.



(a) Grafo de dependencias

	A
1	`10`
2	`2 + 3`
3	`A1 + A2`
4	`sum [A1:A3]`

(b) Contenido de la hoja de cálculo

Figura 3.6: Representación de las dependencias en una hoja de cálculo

El uso de un grafo dirigido permitirá la detección de referencias circulares antes de comenzar la evaluación del contenido de una celda. Para comprobar que no existen referencias circulares simplemente hay que comprobar que no existen ciclos en el grafo.

Además permitirá establecer un orden de evaluación de las celdas utilizando el algoritmo de ordenamiento topológico, una vez se ha comprobado que es un grafo acíclico.

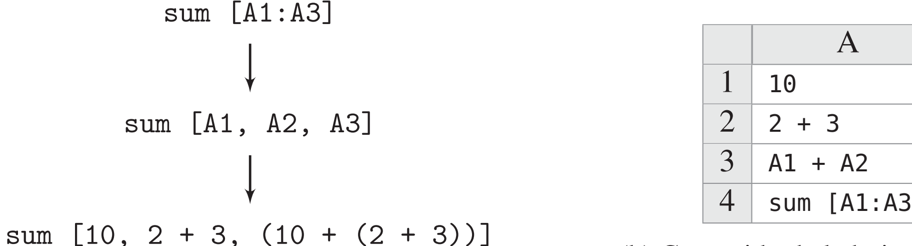
3.5.2 Traducción del contenido de una celda

El proceso de traducción del contenido de una celda consiste en obtener una expresión *Haskell* evaluable de la entrada generada por el usuario.

En el ejemplo ilustrado en la figura 3.7 se parte de la expresión `sum [A1:A3]`, que significa "sumar el contenido de las celdas $A1$ a $A3$", y se obtiene la expresión `sum [10, 2 + 3, (10 + (2 + 3))]` que es una expresión *Haskell* evaluable.

La traducción se llevará a cabo en dos fases:

- En primer lugar se traducirán los rangos escritos por el usuario, obteniendo una expresión con las referencias que describe ese rango. En el ejemplo ilustrado en la figura 3.7a, se traduce el rango `[A1:A3]` a `[A1, A2, A3]`.
- A continuación, se resolverán las referencias entre celdas. Este proceso se realizará utilizando el grafo de dependencias descrito en la sección 3.5.1.



(a) Pasos de la traducción de una expresión

	A
1	10
2	2 + 3
3	A1 + A2
4	sum [A1:A3]

(b) Contenido de la hoja de cálculo

Figura 3.7: Proceso de traducción del contenido de una celda

Como se muestra en la figura 3.7, no sólo hay que resolver las referencias de la celda que se está traduciendo, además se deben resolver las referencias de las celdas de las que depende la misma.

Al finalizar este proceso se obtendrá una expresión *Haskell* evaluable.

El último problema que puede aparecer durante este proceso es que aparezcan celdas sin contenido. En este caso, se ha decidido que al encontrarse una celda vacía se generará un error y se notificará al usuario.

3.5.3 Evaluación del contenido de una celda

La evaluación del contenido de una celda se realizará utilizando un intérprete de *Haskell*. En este proyecto se utilizará el *GHCi*[2]. La evaluación de la expresión obtenida en la figura 3.7a da como resultado:

```
ghci> sum [10, 2 + 3, (10 + (2 + 3))]
30
ghci>
```

Al evaluar una expresión podrá ser incorrecta por varios motivos:

- No estar correctamente escrita.
- No tener un tipo correcto, es decir que evalúe a un valor concreto y que este valor se pueda mostrar por pantalla[3].
- Generar una evaluación infinita.

Todos estos problemas deberán ser detectados por el entorno de ejecución y ser notificados al usuario para que pueda solventarlos.

[2] *Glasgow Haskell Compiler interactive interpreter*
[3] Debe tener una instancia de *Show*

4
Desarrollo

Como se ha explicado en la sección 3.2, la implementación del programa se ha dividido en dos partes, el entorno de ejecución y la interfaz gráfica de usuario. El código del proyecto se encuentra disponible en la plataforma *GitHub* en los siguientes enlaces:

- **Entorno de ejecución:** `https://github.com/edububa/FunCell-runtime/releases/tag/v1.0`
- **Interfaz gráfica de usuario:** `https://github.com/edububa/FunCell-web/releases/tag/v1.0`

En este capítulo se explica el resultado del desarrollo del proyecto. Se tratan los tipos de datos comunes a ambas partes del desarrollo, las funciones específicas de cada parte y su implementación.

4.1 Trabajos previos

En esta sección se tratan distintos prototipos implementados durante el aprendizaje de las tecnologías necesarias para el desarrollo del proyecto.

Todos los ejemplos se encuentran disponibles en: `https://github.com/edububa/TFG/tree/master/code_examples`

El objetivo de este capítulo es mostrar a través de ejemplos sencillos cómo conectar un componente web desarrollado en *PureScript* con un servidor desarrollado en *Haskell* utilizando *WebSockets* como mecanismo de comunicación.

4.1.1 Ejemplo de implementación de un componente web sencillo

En esta sección se va a ilustrar cómo implementar un componente web sencillo usando el *framework* de *PureScript Halogen*. En primer lugar, se definirá el componente web así como su funcionamiento básico. Después, se mostrarán las diferentes funciones que se deben implementar para obtener el componente web. Finalmente, se mostrarán los resultados obtenidos y el funcionamiento del componente web.

Requisitos previos, compilación y ejecución

Para la implementación de este ejemplo se requiere tener una instalación correcta de:

- *PureScript*[1]: lenguaje de programación funcional puro, de evaluación estricta orientado al desarrollo web. Es el lenguaje que se utilizará para implementar el ejemplo.
- El gestor de paquetes `npm`[2], que se utilizará para instalar las dependencias de *JavaScript* necesarias para desarrollar en *PureScript*.
- El gestor de paquetes `bower`: es el gestor de dependencias de *PureScript*, se requerirá para instalar las dependencias necesarias para el componente web.
- La plantilla[3] de proyectos de *Halogen*, sobre la que se implementará el componente web. Contiene las dependencias necesarias para el uso del *framework*.

La compilación del proyecto se llevará a cabo utilizando el mandato:

```
$ pulp build --to dist/app.js
```

Esto generará un fichero de *JavaScript* con el contenido del componente en la carpeta `dist`. Para ejecutar el componente basta con abrir el fichero `dist/index.html`.

Implementación del componente web

El contador[4] va a ser un componente web que almacenará en su estado interno un valor numérico. Este valor numérico se podrá modificar a través de tres operaciones:

- **Incrementar**: esta operación toma el estado actual del contador e incrementa su valor interno una unidad.

[1] Se encuentra disponible en: `http://www.purescript.org`
[2] Se encuentra disponible en: `https://www.npmjs.com/get-npm`
[3] Se encuentra disponible en: `https://github.com/slamdata/purescript-halogen-template.git`
[4] El código fuente de este ejemplo se encuentra disponible en: `https://github.com/edububa/TFG/tree/master/code_examples/halogen-example`

- **Decrementar**: esta operación toma el estado actual del contador y decrementa su valor interno en una unidad.
- **Reinicio**: esta operación devuelve el contador a su valor original, en nuestro caso el valor original será 0.

En este ejemplo se van a permitir valores negativos en el estado de nuestro contador, ya que está se enfocado en ilustrar el uso del *framework*, buscando la mayor simplicidad posible en el mismo.

En primer lugar hay que definir un módulo para el componente web del contador. Este fichero se llamará `Component.purs`. En él se debe definir y exportar el componente web.

```
module Component where

import Halogen as H              -- Halogen
import Halogen.HTML as HH        -- DSL para escribir HTML
import Halogen.HTML.Events as HE -- DSL para manejar eventos

component :: H.Component HH.HTML Query Unit Void Aff
component =
  H.component
    { initialState: const initialState
    , render
    , eval
    , receiver: const Nothing
    }
```

En el código anterior se muestra la implementación del componente web. Se puede apreciar que un componente web necesita un estado inicial (`initialState`), una función de renderizado en pantalla (`render`) que genera el *HTML* que se mostrará en pantalla y una función de evaluación (`eval`).

A continuación, se va a modelizar el estado del contador. Como se ha mencionado anteriormente, el contador no es más que un número entero que se irá incrementando y decrementando. En *PureScript* se puede modelizar como un tipo *record* que contiene un entero. Además, hay que definir los mensajes que harán que el componente cambie de estado. Estos mensajes se modelizarán con un tipo suma que contenga cada mensaje que se puede realizar en el componente web.

```
data Query a = Inc a | Dec a | Zero a

type State = { n :: Int }
```

Como se ha especificado en la definición del componente web iniciamos el estado inicial a 0.

```
initialState :: State
initialState = { n: 0 }
```

En la función de renderizado se utiliza un *DSL* para escribir *HTML* definido en *Halogen*. Este componente consta de tres partes:

- Un título que muestra el nombre del componente. El título es simplemente un campo de texto.

```
HH.text "Counter"
```

- Un texto que muestra el valor actual del contador. Este campo será un texto que obtenga datos del estado del componente.

```
HH.text ("The counter value is: " <> show state.n)
```

- Tres botones para accionar cada una de las tres operaciones del contador. Cada botón debe desencadenar un evento de modificación del estado al ser pulsado.

```
HH.button
    [ HE.onClick (HE.input_ Inc) ]
    [ HH.text "inc" ]
```

En la función de renderizado (`render`) se componen las tres partes anteriores y se introducen en un elemento *HTML*.

```
render :: State -> H.ComponentHTML Query
render state =
  HH.div_
  [ HH.h1_
    [ HH.text "Counter" ]
  , HH.p_
    [ HH.text ("The counter value is: " <> show state.n) ]
  , HH.button
    [ HE.onClick (HE.input_ Inc) ]  -- Operación de incremento
    [ HH.text "inc" ]
  , HH.button
    [ HE.onClick (HE.input_ Dec) ]  -- Operación de decremento
    [ HH.text "dec" ]
  , HH.button
    [ HE.onClick (HE.input_ Zero) ] -- Operación de reinicio
    [ HH.text "0" ]
  ]
```

Finalmente queda implementar la funcionalidad del componente web a través de la función de evaluación de eventos (`eval`). Esta función recibe un mensaje generado al producirse algún evento y realiza distintas acciones. En el caso del contador se ha seleccionado que realice dos acciones:

- La modificación del estado interno del componente web. Para modificar el estado interno del componente web se utilizará la función de orden superior `modify`, definida en *Halogen*. Esta función recibe una función que cambia el estado y la aplica

al estado del componente web.

- La emisión de un mensaje por la consola de *JavaScript*, mostrando el cambio que se ha realizado en el estado del componente. Para realizar este efecto, se debe hacer un *lift* del `log` sobre la mónada `Aff`[5].

```
eval :: Query ~> H.ComponentDSL State Query Void Aff
eval q = do
  let f = case q of              -- Esta es la función que modificará el estado
        Inc  next -> Tuple (\state -> { n: state.n + 1 }) next
        Dec  next -> Tuple (\state -> { n: state.n - 1 }) next
        Zero next -> Tuple (\state -> { n: 0 })           next
  _ <- H.modify $ fst f          -- Se modifica el estado
  state <- H.get                 -- Se obtiene el estado modificado
  _ <- H.liftEffect $ log $ "the state has changed to: " <> show state.n
  pure $ snd f
```

Y en este punto ya se ha finalizado la implementación del contador. Solamente queda definir la función principal que va a ejecutar el componente web. Esta función principal (`main`) carga el cuerpo de la página web y a continuación, ejecuta el componente del contador.

```
main :: Effect Unit
main = HA.runHalogenAff do
  body <- HA.awaitBody
  runUI component unit body
```

En la figura 4.1 se muestra un ejemplo de la ejecución del contador.

4.1.2 Ejemplo de conexión cliente-servidor utilizando *WebSockets*

En esta sección se va a ilustrar el proceso de conexión de componentes web desarrollados en *PureScript* con un servidor en *Haskell*.

En primer lugar, se mostrará la implementación de dos sencillos programas que harán la función de cliente y servidor. La comunicación entre estos dos programas se realizará a través del protocolo de *WebSocket* [12]. A continuación, se enseñará el proceso necesario para serializar/deserializar tipos de datos en *Haskell* para mandarlos a través de *WebSockets*.

Finalmente, se conectará el componente web implementado en la sección 4.1.1 al servidor desarrollado en este ejemplo, la comunicación entre ambas partes se realizará enviando datos serializados en *JSON* a través del *WebSocket*.

[5] La mónada `Aff` es la utilizada por PureScript para realizar efectos asíncronos.

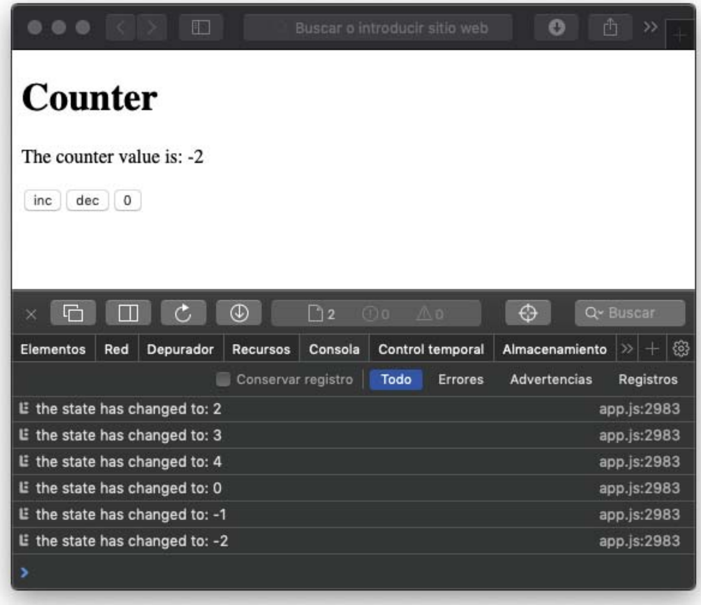


Figura 4.1: Componente web del contador

Requisitos previos, compilación y ejecución

Las herramientas[6] necesarias para compilar y ejecutar este ejemplo son:

- *GHC*, *Glasgow Haskell Compiler*: compilador de *Haskell* más extendido.
- *Stack*: herramienta de gestión de proyectos de *Haskell*.

Lo compilación del proyecto se realizará a través del comando:

```
$ stack build
```

Y la ejecución del servidor se llevará a cabo a través del mandato:

```
$ stack exec server
```

[6] Disponible en: https://www.haskell.org/downloads/

Implementación de un cliente-servidor utilizando *WebSockets*

Este ejemplo[7] va a implementarse completamente en el lenguaje de programación *Haskell*. Se utilizará la biblioteca `websockets`[8] que proporciona las funcionalidades básicas necesarias para implementar programas que utilicen *WebSockets*.

El servidor de este ejemplo consiste en un sencillo programa que responde `OK` a cualquier texto que recibe a través del *WebSocket*. En primer lugar hay que definir el comportamiento del servidor a través de la función `application`.

```
application :: ServerState -> WS.ServerApp
application state pending = do
  conn <- WS.acceptRequest pending -- se inicia la conexión
  putStrLn "Connection started!"
  forever $ do
    msg  <- WS.receiveData conn    -- se espera a recibir datos
    putStrLn $ "[RECEIVED]: " <> (show (msg :: Text))
    WS.sendTextData conn $ ("OK" :: Text) -- se envía el mensaje de
                                          -- confirmación
```

En primer lugar, se establece la conexión al *WebSocket*. En este ejemplo se ha utilizado la función `forever` que repite constantemente la acción que recibe como argumento. Esta acción consiste en recibir un mensaje a través del *WebSocket*, imprimirlo por pantalla y finalmente, envía un mensaje con el texto `OK` a través de la conexión establecida.

El cliente de este ejemplo será un programa de consola que recibirá una línea por consola y la mandará a través del *WebSocket* al servidor.

```
application :: WS.ClientApp ()
application conn = do
  putStrLn "Connected!"
  let loop = do
        line <- T.getLine         -- se obtiene una línea por consola
        unless (T.null line) $ do -- si la línea no está vacía
          WS.sendTextData conn line  -- se envía su contenido
          msg <- WS.receiveData conn -- se espera una respuesta
          putStrLn $ "[RECEIVED]: " <> show (msg :: Text)
          loop                    -- se repite la acción
  loop
  WS.sendClose conn ("Bye!" :: Text)
```

En la función `application` anterior se muestra la implementación del cliente. Como se puede observar, espera una línea a través de la entrada estándar, y si no es vacía, se envía a través de la conexión con el servidor.

[7] El código de este ejemplo se encuentra disponible en: `https://github.com/edububa/TFG/tree/master/code_examples/haskell-websockets`

[8] Disponible en `http://hackage.haskell.org/package/websockets`

4 Desarrollo

A continuación se muestra un ejemplo de la ejecución del cliente y el servidor.

```
Starting Server... Done!
Connection started!
[RECEIVED]: "hola"
```

```
Starting Client... Connected!
hola
[RECEIVED]: "OK"
```

Serialización de tipos de datos de *Haskell* en *JSON*

Para facilitar la comunicación entre cliente y servidor, los mensajes que intercambien se van a serializar en *JSON*. Para ello se ha optado por utilizar la biblioteca `aeson`[9]. Esta biblioteca proporciona un gran rendimiento sobre funciones de serialización y deserialización de *JSON*.

A continuación, se va a definir el tipo de datos de los mensajes que se mandarán a través del *WebSocket* entre el cliente y el servidor. Para que esta implementación se realice de manera automática, se puede utilizar la extensión del compilador `DeriveGeneric` como se muestra en el código a continuación:

```haskell
{-# LANGUAGE DeriveGeneric #-} -- extensión del compilador
{-# LANGUAGE OverloadedStrings #-}
module Lib where

import Data.Aeson (FromJSON, ToJSON, decode, encode)
import Data.Text
import GHC.Generics

data Msg = Msg { content :: Text } deriving (Show, Generic)

instance FromJSON Msg          -- instancia para transformar un JSON
                               -- en un tipo de datos
instance ToJSON Msg            -- instancia para transformar un tipo
                               -- de datos en un JSON
```

Se puede ver que `aeson` utiliza dos *type classes* para tratar con tipos de datos que se pueden codificar en *JSON*:

- `toJSON` que permite serializar un tipo de datos en *JSON*.
- `fromJSON` que permite obtener un tipo de datos de su representación en texto codificado en *JSON*.

[9]Disponible en: `http://hackage.haskell.org/package/aeson`

Para añadir esta funcionalidad al ejemplo anterior simplemente hay que importar los nuevos tipos de datos con sus instancias y utilizar las funciones `encode` y `decode` cuando sea necesario. A continuación se muestra un ejemplo de la ejecución de este ejemplo:

```
Starting Server... Done!
Connection started!
[ENCODED]: "{\"content\":\"hola\"}"
[DECODED]: Msg {content = "hola"}
```

```
Starting Client... Connected!
hola
[ENCODED]: "{\"content\":\"OK\"}"
[DECODED]: Msg {content = "OK"}
```

Conexión de un componente web con un servidor a través de *WebSockets*

En esta última sección del capítulo se unirá el componente web del contador desarrollado en la sección 4.1.1 y el servidor de la sección 4.1.2. Al finalizar esta sección, se tendrá un prototipo[10] sencillo con la misma arquitectura de comunicación que tendrán las hojas de cálculo desarrolladas en este trabajo.

Para conseguir este objetivo, simplemente hay que añadir la capacidad de mandar y recibir mensajes al componente web y añadirle la capacidad de serializar en *JSON* los tipos de datos que se mandan a través de los mensajes.

Serialización de tipos de datos de *PureScript* en *JSON*

La serialización de tipos de datos en *PureScript* se realiza de una forma similar a la mostrada en la sección 4.1.2. Para conseguirlo, se ha utilizado la biblioteca `argonaut`[11], que ofrece unas funcionalidades similares a `aeson`.

En primer lugar hay que definir el nuevo tipo de datos a usar. Después hay que implementar dos instancias de las *type classes* `decodeJson` (equivalente a `fromJSON`) y `encodeJson` (equivalente a `toJSON`).

```
newtype Message = Message { content :: String }

instance decodeJsonMessage :: DecodeJson Message where
  decodeJson json = do
    obj <- decodeJson json
    content <- obj .? "content"
    pure $ Message { content }
```

[10]El código mostrado en esta sección se encuentra disponible en: https://github.com/edububa/TFG/tree/master/code_examples/purescript-ws-json

[11]Disponible en: https://github.com/purescript-contrib/purescript-argonaut

```
instance encodeJsonMessage :: EncodeJson Message where
  encodeJson (Message msg) = "content" := msg.content ~> jsonEmptyObject
```

El código anterior muestra cómo implementar las instancias `decodeJson` y `encodeJson` utilizando `argonaut` en *PureScript*.

Conexión de un componente web con un servidor mediante *WebSockets*

Finalmente, se deben realizar unas pequeñas modificaciones al componente web de la sección 4.1.1 para poder conectarlo a través de un *WebSocket*[12]. Según la guía de *Halogen*[13], hay que implementar tres funciones para que el componente web funcione correctamente:

- `wsSender`: obtiene los mensajes de salida del componente web y los envía a través del *WebSocket*.
- `wsProducer`: una rutina que emite los mensajes que llegan a través del *WebSocket*.
- `wsConsumer`: una rutina que envía mensajes que recibe a través del productor hacia el componente web.

Viendo la función de cada una de las funciones, las realmente importantes y que varían en cada implementación son `wsSender` y `wsConsumer` que son las que envían mensajes y tratan los mensajes que se envían y se reciben.

```
wsSender :: WS.WebSocket -> CR.Consumer Message Aff Unit
wsSender socket = CR.consumer \msg -> do
  case msg of
    OutputMessage msgContents -> -- si es un mensaje de salida se
                                 -- envía a través del socket
      liftEffect $ WS.sendString socket msgContents
  pure Nothing

wsConsumer :: (Query ~> Aff) -> CR.Consumer String Aff Unit
wsConsumer query = CR.consumer \msg -> do
  _ <- H.liftEffect $ log $ "server: " <> msg -- imprime el mensaje
                                               -- recibido
  pure Nothing
```

Una vez añadidas estas funciones quedan dos cambios por hacer, modificar la función `eval` para que genere los mensajes de salida y modificar la función principal para que enrute los mensajes correctamente.

[12] El código de este ejemplo se encuentra disponible en: `https://github.com/edububa/TFG/tree/master/code_examples/purescript-ws-json`

[13] Ejemplo original: `https://github.com/slamdata/purescript-halogen/tree/master/examples/driver-websockets/src`

La función `eval` usa el mismo código que tenía originalmente, pero ahora se ha añadido el proceso de serialización del estado del contador en un *JSON*. Finalmente se genera un mensaje `OutputMessage` que consumirá `wsConsumer` enviándolo al servidor.

```
eval :: Query ~> H.ComponentDSL State Query Message Aff
eval q = do
  let f = case q of
        Inc  next -> Tuple (\state -> { n: state.n + 1 }) next
        Dec  next -> Tuple (\state -> { n: state.n - 1 }) next
        Zero next -> Tuple (\state -> { n: 0 })           next
  _ <- H.modify $ fst f
  state <- H.get
  _ <- H.liftEffect $ log $ "the state has changed to: " <> show state.n
  let msg  = Message { content: (show state.n) } -- se serializa el
      json = stringify <<< encodeJson $ msg      -- mensaje del estado
  _ <- H.liftEffect $ log (show json)
  H.raise $ OutputMessage json  -- se envía el mensaje a wsSender
  pure $ snd f
```

Finalmente hay que hacer unos pequeños ajustes en la función principal del componente. Hay que suscribir `wsSender` a todos los mensajes de salida y conectar el consumidor al productor.

```
main :: Effect Unit
main = do
  connection <- WS.create "ws://127.0.0.1:9160" []
  HA.runHalogenAff do
    body <- HA.awaitBody
    io <- runUI component unit body
    io.subscribe $ wsSender connection -- se suscribe wsSender a todos
                                       -- los mensajes de salida
    -- conectando el consumidor al productor
    CR.runProcess (wsProducer connection CR.$$ wsConsumer io.query)
```

Para ejecutar este ejemplo, en primer lugar debe levantarse el servidor desarrollado en la sección 4.1.2. Una vez levantado, hay que abrir el fichero `dist/index.html`. Después de realizar varias acciones sobre el componente web, se pueden apreciar los resultados obtenidos en la figura 4.2.

```
Starting Server... Done!
Connection started!
[ENCODED]: "{\"content\":\"1\"}"
[DECODED]: Msg {content = "1"}
[ENCODED]: "{\"content\":\"0\"}"
[DECODED]: Msg {content = "0"}
```

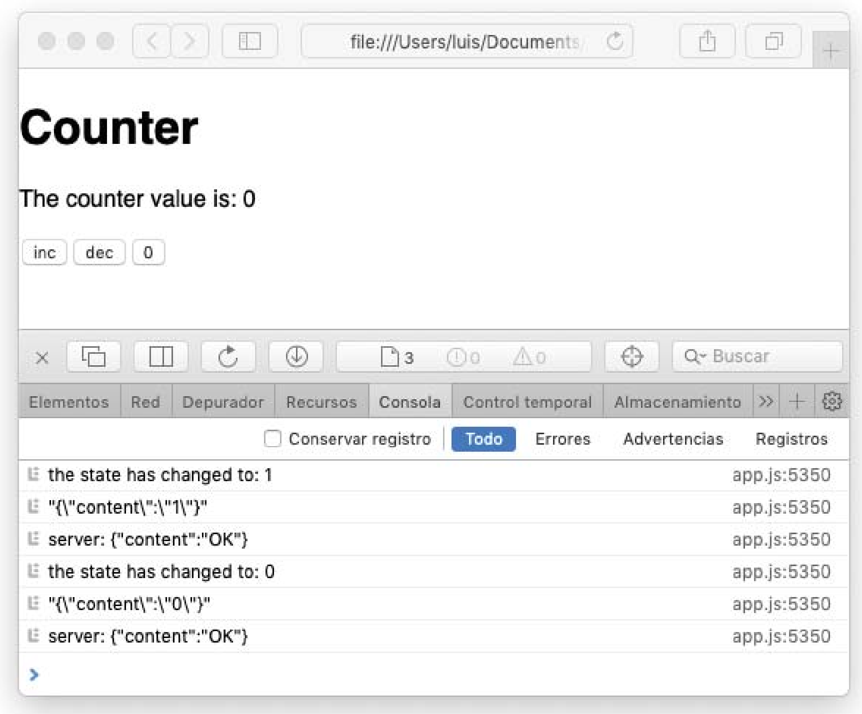


Figura 4.2: Componente web del contador utilizando *WebSockets*

4.2 Interfaz gráfica de usuario

En esta sección se explican los detalles de la implementación de la interfaz gráfica de usuario desarrollada en este proyecto. El código fuente puede encontrarse completo en el apéndice A.2.

La interfaz gráfica se ha implementado como un componente web con *Halogen* siguiendo los pasos mostrados en 4.1

4.2.1 Requisitos previos y compilación

Los requisitos previos para compilar un componente web desarrollado en *PureScript* usando *Halogen* están explicados en la sección 4.1.1. La compilación y ejecución se realizan siguiendo los pasos descritos en esa sección.

4.2.2 Estructura del paquete

En esta sección se explica la estructura del paquete que contiene el código fuente de la interfaz gráfica del usuario.

```
FunCell-web
├── dist
│   ├── app.css
│   ├── app.js
│   └── index.html
├── src
│   ├── Component
│   │   ├── Render
│   │   │   └── Lib.purs
│   │   ├── Communication.purs
│   │   ├── Data.purs
│   │   └── SpreadSheet.purs
│   ├── Data
│   │   ├── Cell
│   │   │   └── Lib.purs
│   │   ├── Cell.purs
│   │   ├── ExternalModule.purs
│   │   └── Messages.purs
│   └── Main.purs
├── bower.json
├── entry.js
└── package.json
```

La carpeta `dist` contiene los ficheros necesarios para abrir la interfaz en un navegador web[14]. En su interior se encuentra:

- `app.css` que contiene la información del estilo de la interfaz web.
- `app.js` que es el fichero resultante de compilar el proyecto. Contiene toda la información de la ejecución que se realizará en la interfaz web.
- `index.html` que es el fichero que debe abrirse para ejecutar correctamente la interfaz web. Al ser abierto carga los dos anteriores.

En la carpeta `src` se encuentra el código fuente del programa. Aquí se encuentran todos los ficheros escritos en *PureScript* que dan funcionalidad a la interfaz web. El código fuente está estructurado:

- La carpeta `Component` que contiene el código fuente del componente web. Los ficheros más importantes de esta carpeta son:
  - `Render/Lib.purs` que contiene las funciones necesarias para renderizar el componente web en pantalla.
  - `Communication.purs` que contiene las funciones para la comunicación con el entorno de ejecución a través del *WebSocket*.
  - `Data.purs` que contiene la información del estado que almacenará el componente web. También contiene los mensajes que se envían internamente cuando se interactúa con el usuario o se reciben resultados del entorno de ejecución.
  - `SpreadSheet.purs` que contiene el estado inicial del componente web y la función de evaluación de los mensajes internos.

[14] Es recomendable el uso de *Chrome* o *Firefox* para visualizar esta interfaz web

- La carpeta `Data` que contiene el código fuente de los datos que se intercambiarán con el entorno de ejecución. Los ficheros principales de esta carpeta son:
  - `Cell.purs` que contiene el tipo de dato de una celda y las instancias necesarias para serializarla a través del *WebSocket*.
  - `Cell/Lib.purs` que contiene funciones que facilitan el uso de celdas y de la estructura de datos de la hoja de cálculo.
  - `ExternalModule.purs` que contiene el tipo de dato que almacena la información sobre el módulo externo y las instancias para serializarlo a través del *WebSocket*.
  - `Messages.purs` que contiene los tipos datos que codifican diferentes mensajes que se intercambian con el servidor.
- El fichero `Main.purs` que contiene el programa principal de la interfaz web.

El proyecto también contiene los ficheros `bower.json` y `package.json` que tienen la información de las dependencias utilizadas en el proyecto.

4.2.3 Estado del componente web

El estado del componente web se ha modelizado como:

```
type State = { spreadSheet    :: SpreadSheet Cell
             , selectedCell   :: Tuple String String
             , externalModule :: String
             , filePath       :: String }
```

En este estado se almacena:

- Una estructura de datos que representa la hoja de cálculo. Esta estructura de datos está implementada con un `Map` que almacena en su interior datos de tipo `Celda`[15].
- Una tupla que representa el contenido y el resultado almacenado en la celda seleccionada.
- Una cadena de caracteres que almacena el contenido del módulo externo.
- Una cadena de caracteres que almacena la ruta del fichero sobre el que se está trabajando.

Los mensajes que se han definido para manejar los eventos generados en el componente web son:

```
data Query a = Update (Tuple Row Col) String a
             | UpdateFocus (Tuple Row Col) a
             | Eval (Tuple Row Col) a
```

[15] Esta representación de celdas es la misma que la mostrada en el entorno de ejecución en la sección 4.3.3

```
           | UpdateResult Cell a
           | UpdateExternalModule String a
           | SendExternalModule a
           | UpdateFilePath String a
           | SaveFile a
           | LoadFile a

data Message = OutputMessage String
```

Los mensajes del tipo `Query` se utilizan para manejar el comportamiento interno del componente web. Los mensajes del tipo `Message` se utilizan para codificar un mensaje entrante a través del *WebSocket*. El funcionamiento de estos mensajes se explica en la sección 4.2.4.

4.2.4 Componentes y eventos

Como se ha propuesto en la sección 3.4.1, para conseguir una experiencia *WYSIWYG* la implementación se ha orientado a eventos, y a través de una comunicación completamente asíncrona con el servidor.

Cada componente de la interfaz tiene un comportamiento dependiendo de distintos eventos:

- **Celdas**: cada celda es una entrada de texto. Reaccionan a los siguientes eventos:
  - Escritura de texto, utiliza el mensaje `Update` y desencadena una acción de actualización del contenido de estructura de datos que almacena la hoja de cálculo.
  - Selección de la celda, utiliza el mensaje `UpdateFocus` y actualiza la información del estado que muestra la celda seleccionada, y el resultado de la misma.
  - Selección de otro campo, utiliza el mensaje `Eval`. Cuando la celda deja de estar seleccionada se procede al envío la información introducida en la celda al servidor.
- **Fórmula** es un contenedor de texto que no se puede editar. Muestra la fórmula completa introducida por el usuario en la celda seleccionada. Su actualización depende de eventos desatados al interactuar con distintas celdas.
- **Resultado** es un contenedor de texto que muestra la información de la última evaluación de la celda seleccionada. Al igual que la fórmula su actualización depende de eventos desatados por las distintas interacciones con las celdas.
- **Módulo Externo** es un área de texto editable que contiene el código del módulo externo. Genera dos eventos:

- Durante la introducción de texto genera el mensaje `UpdateExternalModule` que modifica el estado interno del componente almacenando los datos introducidos en el estado.

- Cuando se selecciona otro campo al dejar de editar el módulo se genera el mensaje `SendExternalModule` que envía la información almacenada al servidor.

- **Ruta de fichero** que es un campo de texto en el que el usuario introduce la ruta del fichero con el que quiere trabajar. Genera un único evento que utiliza el mensaje `UpdateFilePath`. Este evento actualiza el contenido del estado interno con la entrada del usuario.

- **Botones** de guardado y carga, al ser pulsados generan los mensajes de `SaveFile` y `LoadFile` respectivamente. Estos mensajes provocan el envío de mensajes al servidor con la información del fichero que se quiere guardar o cargar y la información de la operación correspondiente.

En la figura 4.3 se muestra el resultado obtenido tras la implementación de esta interfaz.

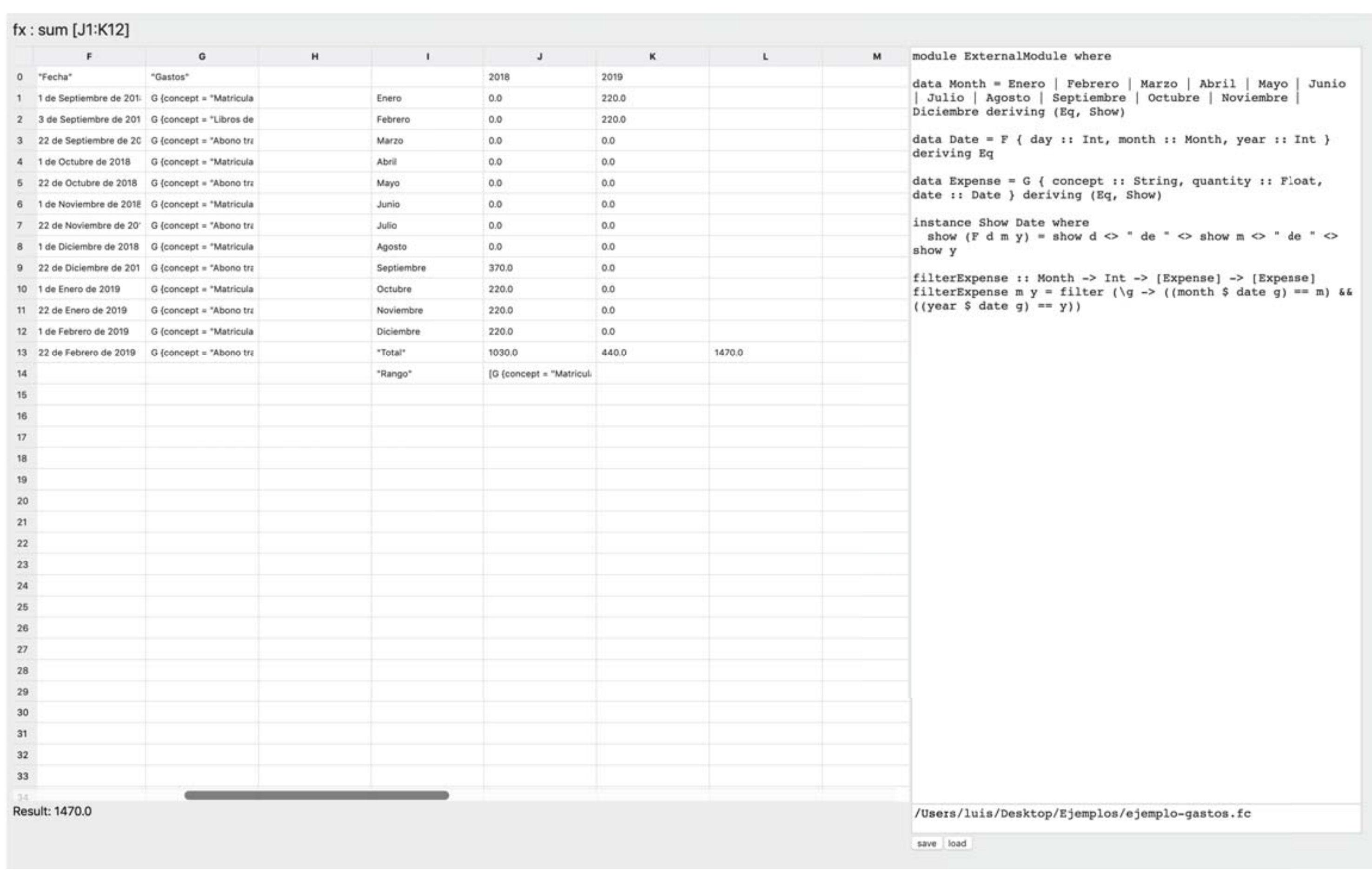


Figura 4.3: Interfaz gráfica de usuario

El último componente importante de la interfaz gráfica de usuario es el consumidor de mensajes generados por el servidor. Esta función puede recibir tres tipos de mensajes:

- El resultado de una celda evaluada por el servidor, que debe actualizar la vista de la celda para mostrar el resultado obtenido.
- El contenido del módulo externo, al cargar una hoja de cálculo de un fichero, se debe actualizar el contenido del módulo externo por el que se almacenó al guardar esa hoja de cálculo.

Para dar esa sensación de actualización automática del contenido de las celdas sin hacer el estado del componente extremadamente complicado se ha optado por implementar una instancia de `Show` sobre el tipo celda que se comporte de la siguiente forma:

```
instance showCell :: Show Cell where
  show (Cell { content: Nothing }) = ""
  show (Cell { evalResult: r, content: (Just c) }) =
    either (const "ERROR") showRight r
    where showRight "" = c
          showRight x  = x
```

- En caso de que el contenido de la celda esté vacío, no muestra nada.
- Si la celda tiene contenido y el resultado de la evaluación está vacío, se mostrará el contenido introducido por el usuario.
- Si la celda tiene un resultado se mostrará "ERROR" si contiene un error y el resultado si este es correcto.

4.3 Entorno de ejecución

En esta sección se explican los detalles de la implementación del entorno de ejecución del prototipo implementado en este proyecto. El código fuente completo puede encontrarse en el apéndice A.1.

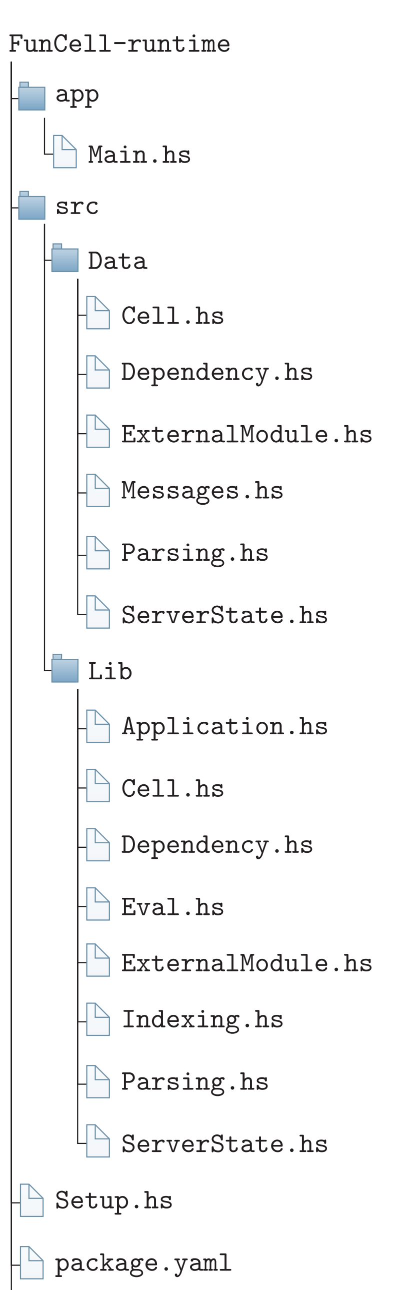
4.3.1 Requisitos previos y compilación

Los requisitos previos y las instrucciones de compilación están explicados en la sección 4.1.2. Las instrucciones de ejecución también están descritos en esta sección.

Para ejecutar la interfaz gráfica de usuario y que la conexión entre ambas partes se realice correctamente, hay que asegurarse de que no hay ningún otro programa utilizando el puerto que utilizará este prototipo.

4.3.2 Estructura del paquete

En esta sección se explica la estructura del paquete que contiene el código fuente del entorno de ejecución de las hojas de cálculo.



La carpeta `app` contiene el programa principal del entorno de ejecución. Este programa se encuentra en el fichero `Main.hs`.

En la carpeta `src` se encuentran los ficheros con el código fuente del programa. Estos ficheros están organizados del siguiente modo:

- La carpeta `Data` que almacena las definiciones de los tipos de datos utilizados por el programa. Los ficheros más importantes de esta carpeta son:
    - `Cell.hs` que contiene la definición del tipo de dato de una celda y las instancias necesarias para enviarla y recibirla a través del *WebSocket*.
    - `Dependency.hs` que contiene la definición del grafo de dependencias.
    - `ExternalModule.hs` que contiene la definición del tipo de dato que almacena el contenido del módulo externo.
    - `Messages.hs` que contiene la definición de los distintos mensajes que se pueden intercambiar con la interfaz gráfica de usuario.
    - `Parsing.hs` que define las expresiones para traducir el contenido de las celdas.
    - `ServerState.hs` que contiene la definición del estado global del servidor y su estado inicial.
- La carpeta `Lib` que contiene las funciones necesarias para facilitar el uso de los tipos de datos definidos en `Data`. Los ficheros principales de esta carpeta son:
    - `Application.hs` que contiene las funciones principales del entorno de ejecución.
    - `Cell.hs`, es una biblioteca que facilita el uso de celdas.

- `Dependency.hs` contiene funciones necesarias para utilizar el grafo de dependencias.
- `Eval.hs` que contiene las funciones de evaluación del contenido de una celda.
- `ExternalModule.hs`, es una biblioteca que facilita el uso del tipo de datos que almacena el módulo externo.
- `Indexing.hs` que contiene funciones de indexado de las celdas de la hoja de cálculo.
- `Parsing.hs` que contiene las funciones de traducción del contenido de una celda.
- `ServerState.hs` que almacena funciones que facilitan la modificación del estado interno del servidor.

El proyecto también contiene los ficheros `Setup.hs`, `package.yaml` y `stack.yaml` que se usan para configurar el proyecto y las dependencias necesarias para su correcto funcionamiento.

4.3.3 Estado interno

El estado interno del servidor se ha modelizado como una tupla que almacena el grafo de dependencias descrito en la sección 3.5.1 y la estructura de datos de hoja de cálculo implementada en el trabajo [1].

Grafo de dependencias

El grafo de dependencias se ha implementado usando la biblioteca `algebraic-graphs` descrita en [14]. Esta biblioteca define una clase para tratar con grafos. Esta clase define las siguientes operaciones:

- `empty` que devuelve un grafo vacío.
- `vertex` que dado un vértice devuelve un grafo.
- `overlay` que superpone el contenido de dos grafos generando un nuevo grafo.
- `connect` que conecta dos grafos.

Este álgebra inicial permite construir cualquier grafo comenzando por cualquier grafo de un único vértice. Estas operaciones básicas permiten la implementación de otras funciones más complejas sobre grafos como la detección de ciclos en grafos dirigidos.

Para la implementación del grafo de dependencias se ha utilizado la estructura de datos definida en esta biblioteca `Algebra.Graph.AdjacencyMap`. Los nodos de este grafo representan los índices de las celdas y las aristas representan las relaciones de dependencias

entre celdas. Sobre este tipo de grafo se han definido distintas operaciones que facilitan el trabajo con dependencias[16].

Hoja de cálculo

La estructura de datos para hojas de cálculo utilizada es una versión especializada de la definida en la biblioteca `haskcell-lib`. Esta estructura de datos se ha especializado utilizando el tipo de datos celda definido en la sección 4.3.3. Además se han definido funciones para facilitar la introducción, extracción y limpiado del contenido de celdas en la hoja de cálculo.

Celdas

La unidad de información más importante en el programa es la celda[17]. La celda se ha definido como un tipo compuesto que contiene:

- `row`, definido como un `Natural` que identifica la fila en la que se encuentra la celda. Con `Natural` se restringe el valor de las filas a un valor entero superior a 0.
- `col`, definido como un `Natural` que identifica la columna en la que se encuentra la celda. Con `Natural` se restringe el valor de las columnas a un valor entero superior a 0.
- `content`, definido por un `Maybe String` que contiene el contenido introducido por el usuario en la celda. En caso de que no contenga información será `Nothing`.
- `evalResult`, definido por el tipo `Either Error String`, que contiene el resultado de evaluar el contenido de la celda. Al evaluarse el contenido de la celda puede devolver un `Error` o un resultado que se almacena en este campo.

Además las celdas son uno de los tipos de datos que se transmiten a través de la red. Para ello se han definido las instancias `ToJSON` y `FromJSON` siguiendo lo explicado en la sección 4.1.2.

4.3.4 Operaciones del servidor

En la figura 3.5 se ha diseñado el funcionamiento general del servidor. Se han definido diferentes operaciones que se concretarán en esta sección.

[16] Su código se encuentra disponible en la sección A.1.2
[17] El código fuente completo de las celdas puede encontrarse en A.1.2

Configuración del programa servidor

La configuración del programa principal se encuentra en la función `main` del módulo `Main`[18].

El proceso de configuración del programa principal es muy sencillo. En primer lugar se genera el fichero donde se almacenará el módulo externo. A continuación se carga la función `application` que tratará los mensajes recibidos a través del *WebSocket*, en el puerto 9160 de la máquina. Esta función puede recibir a través del *WebSocket*:

- Una celda, que desatará la operación de evaluación del contenido de una celda.
- Un mensaje de guardado `Save`, que desatará la operación de guardado de un fichero.
- Un mensaje de carga `Load`, que desatará la operación de carga de un fichero.
- Un módulo externo, que desatará la operación de carga en el contexto del módulo externo.

Todas estas operaciones se han implementado siguiendo el protocolo definido en la sección 3.2.1. Las funciones principales que llevan a cabo estas funciones se encuentran definidas en el módulo `Application`[19].

Guardado del estado en un fichero

El estado del servidor puede guardarse en un fichero en cualquier momento. Cando se recibe un mensaje de guardado, se lee el estado actual, el fichero del módulo externo y se guarda en la ruta pasada en el mensaje.

El guardado en el fichero se realiza codificando el estado en un *JSON*. Para que se pueda realizar correctamente esta operación, todos los tipos de datos que se van a almacenar en el fichero deben contener una instancia *ToJSON*.

Carga del estado de un fichero

La carga del fichero es la operación que permite cargar el estado almacenado previamente en un fichero en la hoja de cálculo. El mensaje que desencadena esta operación contiene una ruta. Los pasos para la carga de un fichero son:

- Comprobar si el fichero existe.
- Si no existe, mandar un mensaje de error de “fichero no encontrado” y terminar la operación.
- Si existe se abre el fichero y se decodifica el *JSON* que lo almacena.

[18]El código fuente completo se encuentra en el apéndice A.1.3
[19]Disponible en el apéndiceA.1.2

- Se tira el estado actual del servidor y se introduce el cargado del fichero.
- Se genera el fichero que contiene el módulo externo con el contenido cargado.
- Se actualiza la interfaz gráfica evaluando cada una de las celdas del nuevo estado.

El último paso de esta carga, hace que el proceso de carga de un fichero no sea demasiado rápido. En un futuro, se deberá evitar el proceso de evaluación completo de la tabla de cálculo al cargar un fichero.

Carga del módulo externo

Al recibir un mensaje con el contenido del módulo externo, este se escribe en el fichero `ExternalModule.hs`. A continuación, se compila para comprobar que no tiene problemas de compilación, se carga en el entorno de ejecución y se procede a volver a evaluar todo el contenido de la hoja de cálculo.

Evaluación de una celda

Cuando llega una celda a través del *WebSocket*, se comienza el proceso de evaluación. El proceso de evaluación de una celda consta de los siguientes pasos:

- Traducción del contenido de la celda, que elimina los rangos y los transforma en sintaxis evaluable por un intérprete de *Haskell*.
- Análisis de dependencias, en este proceso se analiza la expresión resultante, se obtienen las referencias que aparecen en su interior y se comprueba el resultado de introducirlas en el grafo de dependencias. Si se encuentra algún ciclo en este grafo, se devuelve un error y se detiene el proceso de evaluación.
- Si el análisis de dependencias no ha encontrado ningún problema, se añaden estas dependencias al grafo.
- A continuación se actualiza el estado global del programa introduciendo el nuevo grafo de dependencias.
- En este momento se resuelven las referencias de la expresión y se sustituyen por el código correspondiente de las celdas que representan. Si se encuentra algún problema como encontrarse alguna celda vacía, se reporta el error y se detiene la evaluación.
- Si todo ha ido correctamente, se procede a la evaluación de la expresión resultante.

En la implementación actual, el proceso de evaluación puede durar tiempo infinito, y en este caso, el programa se bloquearía dejando de responder a los mensajes recibidos.

5

Resultados

5.1 Ejemplo

En esta sección se muestra cómo implementar el ejemplo inicial explicado en la sección 2.1 en el prototipo desarrollado en este trabajo.

	A	B	C	D	E	G	H
0	**“Curso”**	**“Asignatura”**	**“Créditos”**	**“Nota”**	**“Calificación”**	**“Total”**	60
1	1	“Álgebra lineal”	6	5	Right Aprobado	“Créditos 1”	30
2	1	“Cálculo”	6	7.1	Right Notable	“Créditos 2”	6
3	1	“Lógica”	6	7	Right Notable	“Créditos 3”	9
4	1	“Programación I”	6	9	Right Sobresaliente	“Créditos 4”	15
5	1	“Programación II”	6	8.8	Right Notable		
6	2	“Concurrencia”	3	7.8	Right Notable	**“Media”**	7.95
7	2	“Bases de Datos”	3	5.4	Right Aprobado	“Media 1”	7.38
8	3	*“Middleware”*	3	8	Right Notable	“Media 2”	6.6
9	3	“Sistemas Operativos”	6	8.1	Right Notable	“Media 3”	8.05
10	4	*“Practicum”*	12	9.6	Right Sobresaliente	“Media 4”	9.56
11	4	“Computabilidad”	3	9,4	Right Sobresaliente		

Cuadro 5.1: Ejemplo de hoja de cálculo en el prototipo implementado

En un primer vistazo a la tabla 5.1 ya se aprecian algunas diferencias con el ejemplo original. La diferencia más notable es el uso de comillas para diferenciar la introducción de texto de números. De este modo se evitan problemas como el mencionado en la sección 1.2 en el que un error de entrada de números como texto hizo perder 30.000 £ a un

colegio británico.

La implementación de esta hoja de cálculo ha utilizado la funcionalidad de introducir código a través del módulo externo. En este módulo externo se ha definido el tipo de datos `Calificacion` como el tipo suma:

```
data Calificacion = Suspenso | Aprobado | Notable | Sobresaliente
  deriving (Eq, Show)
```

En el módulo externo también se ha definido una función `calificacion`, que hará la función del condicional para generar las calificaciones sobre las notas en la columna E:

```
calificacion :: (Ord a, Num a) => a -> Either Error Calificacion
calificacion n
  | n >= 0 && n <   5 = Right Suspenso
  | n >= 5 && n <   7 = Right Aprobado
  | n >= 7 && n <   9 = Right Notable
  | n >= 9 && n <= 10 = Right Sobresaliente
  | otherwise         = Left "nota incorrecta"
```

Esta función devolverá la calificación en caso de que la misma esté en el rango correcto y un mensaje de error en caso contrario. La fórmula necesaria para llamar a esta función en las celdas de la columna E es:

```
E1 = calificacion D1
```

Al igual que en el ejemplo original, el análisis de los datos se realiza en las columnas G y H. Las fórmulas utilizadas en estas celdas requieren de un filtrado de celdas por curso, que se define en el módulo externo como:

```
filtroCurso :: Integer -> [(Integer, a)] -> [(Integer, a)]
filtroCurso n = filter (\x -> n == (fst x))
```

La fórmula utilizada para la suma global de todos los créditos es:

```
H1 = sum [C1:C11]
```

La fórmula que suma los créditos filtrándolos por curso consiste en unir en una tupla cada celda de créditos con cada celda de curso, después se hace un filtrado y finalmente se suman los créditos resultantes.

```
H2 = sum $ map snd $ filtroCurso 1 $ zip [A1:A11] [C1:C11]
```

La fórmula para calcular la media global de todos los cursos se realiza combinando las notas con los créditos a través del producto, y la suma dividirla entre el total de créditos, calculado previamente en $H0$:

```
H6 = (sum $ zipWith (*) [C1:C11] [D1:D11]) / H0
```

Finalmente, las fórmulas más complejas en esta tabla son aquellas que realizan la media ponderada por crédito de cada curso. Para ello se vuelve a combinar con un producto los créditos y las notas, el resultado se combina en una tupla con los créditos. Después se filtra

por el curso correspondiente, se suma el resultado y se divide entre el total de créditos de ese curso:

```
H7 = (sum $ map snd $ filtroCurso 1
       $ zip [A1:A11] $ zipWith (*) [C1:C11] [D1:D11])
     / H1
```

Se ha podido reproducir el ejemplo inicial en este prototipo, mejorando la detección del error de introducción de datos de las celdas mostrado en la sección 1.2.

5.2 Ejemplo con constructores de tipos

En esta sección se muestra cómo mejorar la detección de errores sobre los datos de una hoja de cálculo usando constructores de tipos.

En los sistemas de hojas de cálculo tradicionales los tipos de datos se reducen a 4 tipos básicos de datos: números, booleanos, cadenas de texto y errores [1]. Sin embargo, no tienen expresividad suficiente para distinguir tipos más allá de los básicos, por ejemplo, no se pueden distinguir errores, al aplicar funciones a datos distintos representados por números, o por cadenas de caracteres. En el ejemplo explicado en la sección 2.1 existen datos que podrían provocar este tipo de errores por ejemplo los datos de “Curso”, “Créditos” y “Calificación” todos están representados por tipos numéricos.

Para poder distinguirlos, el primer paso es definir los constructores que van a almacenar los tipos numéricos anteriores:

```
newtype Curso = Curso Integer
  deriving (Eq, Num, Ord, Enum, Integral, Real)

newtype Creditos = Creditos Integer
  deriving (Eq, Num, Ord, Enum, Integral, Real)

newtype Nota = Nota Float
  deriving (Eq, Num, Fractional)
```

Estos nuevos tipos utilizarán instancias de los tipos a los que envuelven para facilitar su uso. Estas instancias en este caso son:

- `Eq` que permite que el tipo de datos sea comparado.
- `Num` que aporta las funciones básicas sobre tipos numéricos como la suma la multiplicación...
- `Ord` que permite que los valores del tipo tengan un orden establecido.
- `Enum` que permite el uso de funciones de conjuntos enumerables como “siguiente” y “anterior”.

- `Integral`, `Fractional` y `Real` que permiten el uso de funciones relacionadas con las propiedades de cada conjunto de números.

Para poder derivar automáticamente estas instancias como se hace en este ejemplo es necesario el uso de la extensión del lenguaje `GeneralizedNewtypeDeriving`. Si se cargan estos constructores en un intérprete de *Haskell*, podemos comprobar su funcionamiento:

```
ghci> Curso 1 + Creditos 6

<interactive>:8:11: error:
    • Couldn't match expected type ‘Curso’ with actual type ‘Creditos’
    • In the second argument of ‘(+)’, namely ‘Creditos 6’
      In the expression: Curso 1 + Creditos 6
      In an equation for ‘it’: it = Curso 1 + Creditos 6
ghci> Creditos 6 + Creditos 3
9
ghci> Nota 7 + Nota 8 / Creditos 6 + Creditos 3

<interactive>:30:19: error:
    • Couldn't match expected type ‘Nota’ with actual type ‘Creditos’
    • In the second argument of ‘(/)’, namely ‘Creditos 6’
      In the second argument of ‘(+)’, namely ‘Nota 8 / Creditos 6’
      In the first argument of ‘(+)’, namely
        ‘Nota 7 + Nota 8 / Creditos 6’

<interactive>:30:32: error:
    • Couldn't match expected type ‘Nota’ with actual type ‘Creditos’
    • In the second argument of ‘(+)’, namely ‘Creditos 3’
      In the expression: Nota 7 + Nota 8 / Creditos 6 + Creditos 3
      In an equation for ‘it’:
          it = Nota 7 + Nota 8 / Creditos 6 + Creditos 3
ghci> Nota 7 + Nota 8 / (fromIntegral $ Creditos 6 + Creditos 3)
7.888889
ghci>
```

Se puede ver que no se permiten operaciones entre tipos distintos a no ser que estos sean transformados de forma explícita.

Para aprovechar esto y facilitar la detección de errores, se va a definir un tipo compuesto `Asignatura` que almacenará toda la información relativa a cada una de las asignaturas.

```haskell
data Asignatura = Asignatura { nombre :: String
                             , nota :: Nota
                             , calificacion :: Calificacion
                             , creditos :: Creditos
                             , curso :: Curso } deriving Eq
```

Con esto se podrá comprobar que al construir una asignatura, esta se componga únicamente de datos correctos a nivel de tipos. Ahora las asignaturas se almacenarán como datos en la columna F construyendo el tipo de datos compuesto a partir de los tipos almacenados en las celdas $A1 : D1$ utilizando la fórmula:

```
F1 = Asignatura (A1:D1)
```

Si en algún momento alguno de estos datos no es introducido correctamente, no podrá generarse correctamente el dato compuesto `Asignatura` y se mostrará un error.

El uso de tipos de datos compuestos también facilitan la implementación de funciones en el módulo externo. En este ejemplo se han implementado la función de `filtro` y `mediaPonderada` para simplificar las fórmulas utilizadas en las celdas que analizan datos.

```
filtro :: (Asignatura -> a) -> (a -> Bool) -> [Asignatura] -> [Asignatura]
filtro g p = filter (p . g)

mediaPonderada :: [Asignatura] -> Nota
mediaPonderada as = flip (/) (fromIntegral cr) .
                    foldr ((+) . productoNotaCredito) 0 $
                    as
  where productoNotaCredito a = nota a * fromIntegral (creditos a)
        cr = sum . map creditos $ as
```

Con estas funciones las fórmulas se simplifican notablemente haciendo su significado mucho más claro. Las fórmulas de las celdas que almacenan la suma de créditos podrán escribirse como:

```
H0 = sum . map creditos $ [F1:F11]

H1 = sum . map creditos . filtro curso (== Curso 1) $ [F1:F11]
```

y las fórmulas de las medias ponderadas por crédito serán:

```
H7 = mediaPonderada [F1:F11]

H8 = mediaPonderada . filtro curso (== Curso 1) $ [F1:F11]
```

5.3 Validación de datos

Finalmente, en esta sección se va a mostrar cómo validar datos de una tabla utilizando aplicativos. Esta permitirá comprobar ciertos predicados sobre los datos que se almacenan en la tabla.

En primer lugar, para poder validar datos, es necesario implementar una versión del tipo de datos `Either` en el que su aplicativo almacene los datos de errores en vez de tirarlos.

fx : mediaPonderada . filtro curso (== Curso 4) . asignaturas $ [F1:F11]

I		C	D	E	F	G	H	I
0		"Creditos"	"Nota"	"Calificacion"	"Asignatura"		"Total"	60
1		6	5.0	Aprobado	Algebra lineal; nota: 5.0		"Creditos 1"	30
2		6	7.1	Notable	Calculo; nota: 7.1; calific		"Creditos 2"	6
3		6	7.0	Notable	Logica; nota: 7.0; calific		"Creditos 3"	9
4		6	9.0	Sobresaliente	Programacion I; nota: 9.		"Creditos 4"	15
5		6	8.8	Notable	Programacion II; nota: 8			
6		3	7.8	Notable	Concurrencia; nota: 7.8;		"Media"	7.9500003
7		3	5.4	Aprobado	Bases de Datos; nota: 5		"Media 1"	7.3799996
8		3	8.0	Notable	Middleware; nota: 8.0; c		"Media 2"	6.6000004
9	ivos"	6	8.1	Notable	Sistemas Operativos; nc		"Media 3"	8.066668
10		12	9.6	Sobresaliente	Practicum; nota: 9.6; ca		"Media 4"	mediaPonderada . filtro
11		3	9.4	Sobresaliente	Computabilidad; nota: 9			

Result: 9.56

```
{-# LANGUAGE DeriveFunctor #-}
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
module ExternalModule where

import Data.Either (rights)

-- sinonimos
newtype Validation e r = Validation (Either e r) deriving
(Eq, Functor)
newtype Curso = Curso Integer deriving (Eq, Num, Ord, Enum,
Integral, Real)
newtype Creditos = Creditos Integer deriving (Eq, Num, Ord,
Enum, Integral, Real)
newtype Nota = Nota Float deriving (Eq, Num, Fractional)

type Error = String

-- tipos de datos
data Calificacion = Suspenso | Aprobado | Notable |
Sobresaliente deriving (Eq, Show)
data Asignatura = Asignatura
  { nombre :: String
  , nota :: Nota
  , calificacion :: Calificacion
  , creditos :: Creditos
  , curso :: Curso } deriving Eq

-- validadores
validarNombre :: String -> Validation [Error] String
validarNombre n
  | n /= "" = Validation $ Right n
  | otherwise = Validation $ Left ["nombre incorrecto"]

validarNota :: Nota -> Validation [Error] Calificacion
validarNota (Nota n)
  | n >= 0 && n <   5 = Validation $ Right Suspenso
  | n >= 5 && n <   7 = Validation $ Right Aprobado
  | n >= 7 && n <   9 = Validation $ Right Notable
  | n >= 0 && n <= 10 = Validation $ Right Sobresaliente
  | otherwise = Validation $ Left ["nota incorrecta"]

validarCurso :: Curso -> Validation [Error] Curso
validarCurso (Curso n)
  | n >= 1 && n <= 4 = Validation $ Right $ Curso n
  | otherwise = Validation $ Left ["curso incorrecto"]

validarCreditos :: Creditos -> Validation [Error] Creditos
validarCreditos (Creditos n)
  | n > 0 = Validation $ Right $ Creditos n
  | otherwise = Validation $ Left ["creditos incorrectos"]

validarAsignatura :: Curso -> String -> Creditos -> Nota
                  -> Validation [Error] Asignatura
validarAsignatura c n cr nt = pure Asignatura
                              <*> validarNombre n
```

/Users/luis/Desktop/Ejemplos/ejemplo-validacion.fc

save load

Figura 5.1: Hoja de cálculo con validadores

Para ello se ha declarado el tipo de datos `Validation` y su instancia de aplicativo correspondiente:

```haskell
newtype Validation e r = Validation (Either e r)
  deriving (Eq, Functor)

instance Monoid m => Applicative (Validation m) where
  pure = Validation . pure
  Validation (Left x) <*> Validation (Left y) = Validation . Left $ x <> y
  Validation f <*> Validation r = Validation $ f <*> r
```

En la nueva instancia de aplicativo, es necesario que el tipo que almacena los errores contenga una instancia de monoide para que funcione correctamente. Además para derivar automáticamente la instancia `Functor` es necesaria la extensión del lenguaje `DeriveFunctor`. Para implementar los validadores, primero debemos definir cada uno de los predicados que deseamos validar para cada uno de los datos que forman el tipo de datos compuesto `Asignatura`.

Un nombre de una asignatura no podrá ser vacío:

```haskell
validarNombre :: String -> Validation [Error] String
validarNombre n
  | n /= "" = Validation $ Right n
  | otherwise = Validation $ Left ["nombre incorrecto"]
```

Una nota correcta deberá tener un valor superior o igual a 0 e inferior o igual a 10:

```haskell
validarNota :: Nota -> Validation [Error] Calificacion
validarNota (Nota n)
  | n >= 0 && n <   5 = Validation $ Right Suspenso
  | n >= 5 && n <   7 = Validation $ Right Aprobado
  | n >= 7 && n <   9 = Validation $ Right Notable
  | n >= 0 && n <= 10 = Validation $ Right Sobresaliente
  | otherwise = Validation $ Left ["nota incorrecta"]
```

Un curso correcto deberá tener un valor superior o igual a 1 e inferior o igual a 4:

```haskell
validarCurso :: Curso -> Validation [Error] Curso
validarCurso (Curso n)
  | n >= 1 && n <= 4 = Validation $ Right $ Curso n
  | otherwise = Validation $ Left ["curso incorrecto"]
```

El número de créditos de una asignatura deberá ser superior a 0:

```haskell
validarCreditos :: Creditos -> Validation [Error] Creditos
validarCreditos (Creditos n)
  | n > 0 = Validation $ Right $ Creditos n
  | otherwise = Validation $ Left ["creditos incorrectos"]
```

Finalmente, se combinan todos estos validadores para construir una asignatura:

```haskell
validarAsignatura :: Curso -> String -> Creditos -> Nota
                  -> Validation [Error] Asignatura
validarAsignatura c n cr nt = pure Asignatura
                              <*> validarNombre n
                              <*> nt'
                              <*> validarNota nt
                              <*> validarCreditos cr
                              <*> validarCurso c
  where nt' = Validation $ Right nt
```

El funcionamiento de estos validadores es el siguiente:

```
ghci> validarAsignatura (Curso 5) "" (Creditos $ -10) (Nota 20)
Validation (Left ["nombre incorrecto","nota incorrecta","creditos incorrectos"
,"curso incorrecto"])
ghci> validarAsignatura (Curso 1) "Algebra" (Creditos 6) (Nota 5)
Validation (Right (Asignatura {nombre = "Algebra", nota = 5.0, calificacion =
```

```
Aprobado, creditos = 6, curso = 1}))
ghci>
```

Ahora las asignaturas en el módulo externo se construirán validándolas previamente:

```
F1 = validarAsignatura (A1:D1)
```

Para simplificar el uso de los tipos validados, se ha implementado una función `toEither` que extrae el tipo `Either` del interior de `Validation`. También se ha implementado una función `asignaturas` que extrae las asignaturas correctamente validadas.

```
toEither :: Validation e r -> Either e r
toEither (Validation x) = x

asignaturas :: [Validation [Error] Asignatura] -> [Asignatura]
asignaturas = rights . map toEither
```

Las fórmulas que suman créditos y calculan las medias serán exactamente iguales a las mostradas en la sección 5.2, pero extrayendo las asignaturas correctamente validadas:

```
H0 = sum . map creditos . asignaturas $ [F1:F11]

H1 = sum . map creditos . filtro curso (== Curso 1) . asignaturas $ [F1:F11]

H6 = mediaPonderada . asignaturas $ [F1:F11]

H7 = mediaPonderada . filtro curso (== Curso 1) . asignaturas $ [F1:F11]
```

6

Conclusiones

Se ha implementado un prototipo de entorno de ejecución de hojas de cálculo basadas en programación funcional. El prototipo final es capaz de evaluar fórmulas escritas en un subconjunto de *Haskell*, extendido con una sintaxis para el uso de referencias entre celdas y operaciones de rangos.

En el capítulo 2 se ha realizado un análisis detallado de un ejemplo implementado en un sistema de hojas de cálculo tradicional. De este ejemplo se han extraído las funcionalidades básicas que debía cumplir el prototipo, debiendo ser capaz de replicar ese ejemplo.

En el capítulo 3 se han mostrado las diferentes decisiones de diseño que se han tomado sobre el funcionamiento de las hojas de cálculo, buscando la experiencia *WYSIWYG*. Además, se han explicado las diferentes decisiones de diseño referentes al lenguaje de fórmulas utilizado, la traducción de rangos y referencias y el comportamiento global del programa. También se han tomado decisiones sobre cómo tratar las celdas vacías y otro tipo de decisiones, como la detección de errores. En el capítulo 4 se han resumido las diferentes decisiones sobre la implementación concreta del diseño original, explicando el funcionamiento de las tecnologías utilizadas.

Finalmente, en el capítulo 5 se ha conseguido implementar el ejemplo presentado en el capítulo 2 mostrando ventajas básicas que presenta este prototipo en la detección de errores. Después, este ejemplo se ha implementado explotando al máximo las ventajas del sistema de tipos ofrecido por el entorno de ejecución presentado, facilitando la detección prematura de más tipos de errores mediante el uso de constructores de tipos y aplicativos para la validación de datos.

Por tanto, el prototipo obtenido cuenta con la facilidad de uso presentada por los sistemas de hojas de cálculo tradicionales aunque cambia radicalmente la sintaxis utilizada por las fórmulas en las celdas, pero además consigue aumentar enormemente las posibilidades de

detección de errores para usuarios más avanzados exprimiendo al máximo el sistema de tipos de *Haskell*.

6.1 Trabajos futuros

El prototipo final está abierto a distintas mejoras que, por falta de tiempo, no han podido llevarse a cabo. Una de ellas, es el soporte de todos los tipos de rangos, no están implementados los rangos por filas o por columnas. Actualmente, a pesar de contar con un encapsulado de la interfaz gráfica utilizando *Electron*, el entorno de ejecución requiere de la instalación completa del entorno *Haskell* y la compilación manual del prototipo. Y finalmente, habría que mejorar la detección de expresiones que generan evaluaciones infinitas o introducir un *timeout*.

En trabajos futuros podría integrarse con la capacidad de hacer *Property based testing* integrando este entorno de ejecución con las capacidades de *testing* presentadas en el proyecto de fin de carrera [1]. Además, para mejorar el sistema de tipos con el que cuenta actualmente el entorno de ejecución, podría extenderse con tipos refinados, tipos dependientes, sistemas de tipos líquidos o incluso con verificadores.

Bibliografía

[1] I. Ballesteros, *Diseño de una librería para el apoyo a hojas de cálculo funcionales*. Escuela Técnica Superior de Ingenieros Informáticos, 2019.

[2] R. R. Panko, «What We Know About Spreadsheet Errors», *J. End User Comput.*, vol. 10, n.º 2, págs. 15-21, mayo de 1998, ISSN: 1063-2239. dirección: http://dl.acm.org/citation.cfm?id=287893.287899.

[3] R. Panko, «What We Don't Know About Spreadsheet Errors Today: The Facts, Why We Don't Believe Them, and What We Need to Do», *CoRR*, vol. abs/1602.02601, 2016. arXiv: 1602.02601. dirección: http://arxiv.org/abs/1602.02601.

[4] S. G. Powell, K. R. Baker y B. Lawson, «A critical review of the literature on spreadsheet errors», *Decision Support Systems*, vol. 46, n.º 1, págs. 128-138, 2008, ISSN: 0167-9236. DOI: https://doi.org/10.1016/j.dss.2008.06.001. dirección: http://www.sciencedirect.com/science/article/pii/S0167923608001127.

[5] B. Ronen, M. A. Palley y H. C. Lucas Jr., «Spreadsheet Analysis and Design», *Commun. ACM*, vol. 32, n.º 1, págs. 84-93, ene. de 1989, ISSN: 0001-0782. DOI: 10.1145/63238.63244. dirección: http://doi.acm.org/10.1145/63238.63244.

[6] D. W. Barowy, E. D. Berger y B. Zorn, «ExceLint: Automatically Finding Spreadsheet Formula Errors», *Proc. ACM Program. Lang.*, vol. 2, n.º OOPSLA, 148:1-148:26, oct. de 2018, ISSN: 2475-1421. DOI: 10.1145/3276518. dirección: http://doi.acm.org/10.1145/3276518.

[7] B. Ray, D. Posnett, P. Devanbu y V. Filkov, «A Large-scale Study of Programming Languages and Code Quality in GitHub», *Commun. ACM*, vol. 60, n.º 10, págs. 91-100, sep. de 2017, ISSN: 0001-0782. DOI: 10.1145/3126905. dirección: http://doi.acm.org/10.1145/3126905.

[8] M. J. Foley. (2010). About that 1 billion Microsoft Office figure ..., dirección: https://www.zdnet.com/article/about-that-1-billion-microsoft-office-figure/ (visitado 26-06-2019).

[9] EuSpRIG. (2019). Original Horror Stories, dirección: http://www.eusprig.org/stories.htm (visitado 26-06-2019).

[10] M. R. Zynda, «The First Killer App: A History of Spreadsheets», *interactions*, vol. 20, n.º 5, págs. 68-72, sep. de 2013, ISSN: 1072-5520. DOI: 10.1145/2509224. dirección: http://doi.acm.org/10.1145/2509224.

[11] P. Freeman, *PureScript by example*.

[12] I. Fette y A. Melnikov, «The WebSocket Protocol», RFC Editor, RFC 6455, dic. de 2011, http://www.rfc-editor.org/rfc/rfc6455.txt. dirección: http://www.rfc-editor.org/rfc/rfc6455.txt.

[13] T. Bray, «The JavaScript Object Notation (JSON) Data Interchange Format», RFC Editor, STD 90, dic. de 2017.

[14] A. Mokhov, «Algebraic Graphs with Class (Functional Pearl)», *SIGPLAN Not.*, vol. 52, n.º 10, págs. 2-13, sep. de 2017, ISSN: 0362-1340. DOI: 10.1145/3156695.3122956. dirección: http://doi.acm.org/10.1145/3156695.3122956.

A

Código fuente

A.1 Entorno de ejecución

A.1.1 Paquete `Data`

Fichero `Cell.hs`

```
{-| This module contains the type definitions and instances to deal with
  Cells. -}
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE StandaloneDeriving #-}
{-# LANGUAGE OverloadedStrings #-}
module Data.Cell
  ( -- * Type synonyms
    Row
  , Col
  , Index
  , Error
    -- * Data type
  , Cell(..)
  ) where

-- external imports
import Data.Aeson
import Data.SpreadSheet
```

```haskell
import GHC.Generics
import Numeric.Natural

-- Lib data types
type Row = Natural
type Col = Natural
type Error = String
type Index = (Row, Col)

data Cell = Cell { row :: Row
                 , col :: Col
                 , content :: Maybe String
                 , evalResult :: Either Error String } deriving (Generic, Read)

instance Show Cell where
  show (Cell r c cont res) = "(" <> show r <> ", " <> show c <> "): " <>
                             show cont <> " " <> show res

instance FromJSON Cell where
  parseJSON = withObject "cell" $ \o -> do
    r  <- o .: "row"
    c  <- o .: "col"
    ct <- o .: "content"
    return $ Cell r c ct (Right "")

instance ToJSON Cell

deriving instance Generic (SpreadSheet a)

instance ToJSON a => ToJSON (SpreadSheet a)

instance FromJSON a => FromJSON (SpreadSheet a)
```

Fichero `Dependency.hs`

```haskell
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE StandaloneDeriving #-}
{-| This module contains the definition of the Dependency data type. -}
module Data.Dependency where

-- external imports
import Algebra.Graph.AdjacencyMap
import Algebra.Graph.AdjacencyMap.Internal
```

```haskell
import Data.Aeson
import GHC.Generics
-- internal imports
import Data.Cell

{-| 'Dependencies' will be stored in a directed graph. -}
type Dependencies = AdjacencyMap Index

{-| The 'empty' value of an empty dependency graph. -}
empty :: Dependencies
empty = Algebra.Graph.AdjacencyMap.empty

deriving instance Generic (AdjacencyMap a)

instance (ToJSON a, ToJSONKey a) => ToJSON (AdjacencyMap a)

instance (Ord a, FromJSON a, FromJSONKey a) => FromJSON (AdjacencyMap a)
```

Fichero `ExternalModule.hs`

```haskell
{-| This module contains the definition of the ExternalModule data type
  and some useful instances. -}
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE OverloadedStrings #-}
module Data.ExternalModule where

-- external imports
import Data.Aeson
import GHC.Generics

data ExternalModule = ExternalModule String deriving (Show, Generic)

instance FromJSON ExternalModule where
  parseJSON = withObject "externalModule" $ \o -> do
    text <- o .: "text"
    return $ ExternalModule text

instance ToJSON ExternalModule where
  toJSON (ExternalModule e) =
    object [ "text" .= e ]
```

A Código fuente

Fichero `Messages.hs`

```haskell
{-| -}
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE OverloadedStrings #-}
module Data.Messages where

-- external imports
import Data.Aeson
import GHC.Generics

data Save = Save String deriving (Show, Generic)
data Load = Load String deriving (Show, Generic)

instance FromJSON Save where
  parseJSON = withObject "message"  $ \o -> do
    path <- o .: "save"
    return $ Save path

instance FromJSON Load where
  parseJSON = withObject "message" $ \o -> do
    path <- o .: "load"
    return $ Load path
```

Fichero `Parsing.hs`

```haskell
module Data.Parsing where

-- external imports
import Text.Regex

{-| The 'rangeRegex' value defines a regex that matches ranges of
  references. For example: (A0:B0). -}
rangeRegex :: Regex
rangeRegex = mkRegex "[(][A-Z]+[0-9]+[0-9]*[:][A-Z]+[0-9]+[0-9]*[)]"

{-| The 'listRegex' value defines a regex that matches lists of
  references. For example: [A0:B2]. -}
listRegex :: Regex
listRegex = mkRegex "[[][A-Z]+[0-9]+[0-9]*[:][A-Z]+[0-9]+[0-9]*[]]"

{-| The 'referencesRegex' value defines a regex that matches references. -}
referencesRegex :: Regex
```

```
referencesRegex = mkRegex "[A-Z]+[0-9]+[0-9]*"

{-| The 'columnRegex' value defines a regex that matches the column in a
    reference. -}
columnRegex :: Regex
columnRegex = mkRegex "[A-Z]+"

{-| The 'rowRegex' value defines a regex that matches the row in a
    reference. -}
rowRegex :: Regex
rowRegex = mkRegex "[0-9]+[0-9]*"
```

Fichero `ServerState.hs`

```
{-| This module contains the definition of the ServerState used by the
  web server.-}
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE OverloadedStrings #-}
module Data.ServerState where

-- internal imports
import Data.Cell
import Data.SpreadSheet as SS
import Data.Dependency as Dep

type ServerState = (SpreadSheet Cell, Dependencies)

{-| The value 'newServerstate' contains the initial value of the state
  of the server. -}
newServerState :: ServerState
newServerState = (SS.empty, Dep.empty)
```

A.1.2 Paquete `Lib`

Fichero `Application.hs`

```
{-| This module contains a library of functions used by the main
  application of the project -}
module Lib.Application where

-- external imports
import Control.Concurrent
```

A Código fuente

```haskell
import Control.Monad
import Control.Monad.IO.Class
import Control.Monad.Trans.Except
import Control.Monad.Trans.Maybe
import Data.Aeson
import Data.SpreadSheet (toListValues)
import qualified Network.WebSockets as WS
-- internal imports
import Data.Cell
import Data.ExternalModule
import Data.ServerState
import Lib.Cell
import Lib.Dependency
import Lib.Eval
import Lib.ExternalModule
import Lib.Indexing
import Lib.Parsing
import Lib.ServerState
import System.Directory

-- | 'evalCell' evaluates the content of the received cell and sends
-- it back to the server.
evalCell :: MVar ServerState -> WS.Connection -> Cell -> IO ()
evalCell state conn cell = do
  res <- runExceptT $ evalContent state cell
  let cell' = cell { evalResult = res }
  send conn cell'
  modifyMVar_ state $ updateCell cell'

-- | 'save' saves the actual state of the server in the received path.
save :: MVar ServerState -> String -> IO ()
save state path = do
  s <- readMVar state
  file <- readFile "ExternalModule.hs" -- unsafe
  encodeFile path (s, ExternalModule file)
  putStrLn $ "[INFO]: saved file: '" <> path  <> "'"

-- | 'load' loads the state saved in the file received as argument.
load :: MVar ServerState -> WS.Connection -> String -> ExceptT Error IO ()
load state conn path = do
  exists <- liftIO $ doesFileExist path
  when (not exists) $ liftIO $ putStrLn
    $ "[ERROR]: file: '" <> path <> "' not found."
  guard exists
```

```haskell
  file <- liftIO (eitherDecodeFileStrict' path
                :: IO (Either Error (ServerState, ExternalModule)))
  s    <- except file
  liftIO $ modifyMVar_ state $ \_ -> return $ fst s
  saveAndLoadExternalModule $ snd s
  liftIO $ send conn $ snd s
  liftIO $ mapM_ (send conn) (toListValues $ fst . fst $ s)
  maybeToExceptT "" $ updateSpreadSheet state conn
  liftIO $ putStrLn $ "[INFO]: loaded file: '" <> path <> "'"

-- | 'updateDependents' evaluates all the dependent cells of the
-- argument cell.
updateDependents :: MVar ServerState -> WS.Connection -> Cell -> IO ()
updateDependents state conn cell = do
  (ss, deps) <- readMVar state
  let refs = getDependents (getIndex cell) deps
  mapM_ (evalCell state conn . flip getCell ss) refs

-- | 'updateSpreadSheet' evaluates all the cells with content in the
-- spreadsheet. Follows the topological order of the dependency graph.
updateSpreadSheet :: MVar ServerState -> WS.Connection -> MaybeT IO ()
updateSpreadSheet state conn = do
  (ss, ds) <- liftIO $ readMVar state
  is <- MaybeT . return $ getOrder ds
  liftIO . mapM_ (evalCell state conn) . map (flip getCell ss) $ reverse is

-- | 'evalContent' evaluates the content of the given cell. First
-- analyzes its dependencies, if no problems are found, evaluates the
-- content of the cell.
evalContent :: MVar ServerState -> Cell -> ExceptT Error IO String
evalContent _          (Cell { content = Nothing }) = return ""
evalContent state cell@(Cell { content = Just c  }) = do
  desugaredContent <- except $ desugarContent c
  let cell' = cell { content = Just desugaredContent }
  deps <- analyzeDependencies state cell'
  liftIO $ updateState state cell' deps
  (ss, ds) <- liftIO $ readMVar state
  let dependencies = getDependencies (fst deps) ds
  solvedContent <- except $ solveDependencies ss dependencies desugaredContent
  liftIO $ putStrLn $ "[EVAL]: content - " <> solvedContent
  result <- eval solvedContent
  liftIO $ putStrLn $ "[EVAL]: result  - " <> result
  return result
```

A Código fuente

```haskell
-- | 'analyzeDependencies' looks for circular references.
analyzeDependencies :: MVar ServerState -> Cell -> ExceptT Error IO (Index, [Index])
analyzeDependencies _     cell@(Cell { content = Nothing }) = pure (getIndex cell, [])
analyzeDependencies state cell@(Cell { content = Just c  }) = do
  (_, deps) <- liftIO $ readMVar state
  let i    = getIndex cell
      refs = parseReferences c
  except $
    if circularDependencies $ addDependencies i refs deps
    then Left  ("Circular dependencies found")
    else Right (i, refs)

-- | 'send' encodes and sends data that can be converted to
-- JSON through an open WebSocket connection.
send :: ToJSON a => WS.Connection -> a -> IO ()
send conn = WS.sendTextData conn . encode
```

Fichero `Cell.hs`

```haskell
-- | This module contains some useful functions to deal with @Cell@
-- and @SpreadSheet@
module Lib.Cell where

-- external imports
import Data.SpreadSheet (SpreadSheet)
import qualified Data.SpreadSheet as SpreadSheet
-- internal imports
import Data.Cell
import Lib.Indexing

-- | The 'empty' value of a SpreadSheet of Cells. -}
empty :: SpreadSheet Cell
empty = SpreadSheet.empty

-- | 'emptyCell' is a function that creates an empty cell in a given
-- index.
--
-- >>> emptyCell (0,0)
-- (0, 0): Nothing Right ""
emptyCell :: Index -> Cell
emptyCell (r, c) = Cell r c Nothing (Right "")

-- | 'addCell' inserts a cell in a @SpreadSheet@.
```

```
--
-- >>> c = Cell 0 0 (Just "1 + 2") (Right "3")
-- >>> addCell c empty
-- Range (0,0) (0,0)
-- fromList [((0,0),(0, 0): Just "1 + 2" Right "3")]
addCell :: Cell -> SpreadSheet Cell -> SpreadSheet Cell
addCell c = SpreadSheet.put (getIndex c) (const c)

-- | 'getCell' returns a @Cell@ in the index of a given @SpreadSheet@.
getCell :: Index -> SpreadSheet Cell -> Cell
getCell i = maybe (emptyCell i) id . SpreadSheet.get i

-- | 'clearCell' deletes the content of the cell in a given index of a
-- @SpreadSheet@
--
-- >>> c = Cell 0 0 (Just "1 + 2") (Right "3")
-- >>> clearCell (0,0) $ addCell c empty
-- Range (0,0) (0,0)
-- fromList [((0,0),(0, 0): Nothing Right "")]
clearCell :: Index -> SpreadSheet Cell -> SpreadSheet Cell
clearCell = addCell . emptyCell
```

Fichero `Dependency.hs`

```
-- | This module contains the data types and functions to deal with
-- dependencies inside a @SpreadSheet@.
module Lib.Dependency
  ( -- * Dependency Graph Functions
    addDependency
  , addDependencies
  , resetDependency
  , removeDependency
  , updateDependency
    -- * Dependency Analysis Functions
  , getDependencies
  , getDependents
  , circularDependencies
  , getOrder
  ) where

-- external imports
import Algebra.Graph.AdjacencyMap
import Algebra.Graph.AdjacencyMap.Algorithm
```

A Código fuente

```haskell
-- internal imports
import Data.Cell
import Data.Dependency
import Data.Set (toList)

-- | 'addDependency' introduces a new dependency to the graph.
--
-- >>> addDependency (0,0) (1,1) Data.Dependency.empty
-- edge (0,0) (1,1)
-- >>> addDependency (0,0) (0,1) $ addDependency (0,0) (1,1) Data.Dependency.empty
-- edges [((0,0),(0,1)),((0,0),(1,1))]
addDependency :: Index -> Index -> Dependencies -> Dependencies
addDependency from to = overlay $ edge from to

-- | 'addDependencies' introduces dependencies between the first index
-- and a list of indices
--
-- >>> let x = addDependency (0,0) (0,1) $ addDependency (0,0) (1,1) Data.Dependency.empty
-- >>> let y = addDependencies (0,0) [(0,1), (1,1)] Data.Dependency.empty
-- >>> x
-- edges [((0,0),(0,1)),((0,0),(1,1))]
-- >>> y
-- edges [((0,0),(0,1)),((0,0),(1,1))]
-- >>> x == y
-- True op
addDependencies :: Index -> [Index] -> Dependencies -> Dependencies
addDependencies from tos ds = foldr (addDependency from) ds tos

-- | 'resetDependency' deletes the vertex of a given index in the
-- graph, this way resets all its dependencies.
--
-- >>> let x = addDependencies (0,0) [(0,1), (1,1)] Data.Dependency.empty
-- >>> resetDependency (0,0) x
-- vertices [(0,1),(1,1)]
resetDependency :: Index -> Dependencies -> Dependencies
resetDependency = removeVertex

-- | 'removeDependency' deletes the edge representing a dependency in
-- the graph.
--
-- >>> let x = addDependencies (0,0) [(0,1), (1,1)] Data.Dependency.empty
-- >>> removeDependency (0,0) (0,1) x
-- overlay (vertex (0,1)) (edge (0,0) (1,1))
removeDependency :: Index -> Index -> Dependencies -> Dependencies
```

```haskell
removeDependency = removeEdge

-- | 'updateDependency' returns an updated dependency graph, first
-- removes the edges of then Index that will be updated, after that
-- the new edges are added.
--
-- >>> let x = addDependencies (0,0) [(0,1), (1,1)] Data.Dependency.empty
-- >>> updateDependency (0,0) [(1,2), (3,3)] x
-- overlay (vertices [(0,1),(1,1)]) (edges [((0,0),(1,2)),((0,0),(3,3))])
updateDependency :: Index -> [Index] -> Dependencies -> Dependencies
updateDependency from []  ds = overlay (vertex from) ds
updateDependency from tos ds = addDependencies from tos .
                               foldr (removeDependency from) ds .
                               toList . postSet from $ ds

-- | 'getOrder' returns the topological sort of the graph.
getOrder :: Dependencies -> Maybe [Index]
getOrder = topSort

-- | 'getDependencies' returns a list with the dependencies reachable
-- from a given index.
getDependencies :: Index -> Dependencies -> [Index]
getDependencies = reachable

-- | 'getDependents' returns a list with the dependent indices of a
-- given index.
--
-- >>> let x = addDependencies (0,0) [(0,1), (1,1)] Data.Dependency.empty
-- >>> getDependencies (0,0) x
-- [(0,0),(0,1),(1,1)]
getDependents :: Index -> Dependencies -> [Index]
getDependents from deps = pre <> pre'
  where pre = toList <$> preSet from $ deps
        pre' = concat . map (\x -> getDependents x deps) $ pre

-- | 'circularDependencies' is a predicate that checks if a given
-- graph has circular dependencies.
--
-- let x = addDependencies (0,0) [(0,1), (1,1)] Data.Dependency.empty
-- >>> x
-- edges [((0,0),(0,1)),((0,0),(1,1))]
-- >>> circularDependencies x
-- False
-- >>> circularDependencies $ addDependency (0,0) (0,0) x
```

A Código fuente

```haskell
-- True
circularDependencies :: Dependencies -> Bool
circularDependencies = not . isAcyclic
```

Fichero `Eval.hs`

```haskell
-- | This module contains functions to handle the evaluation of the
-- content of cells.
module Lib.Eval
  ( -- * Evaluation Functions
    context
  , eval
    -- * Dependency Solving Functions
  , solveDependencies
    -- ** Auxiliar Functions
  , applyValues
  , applyValue
  , cellToIndexAndVal
  ) where

-- external imports
import Control.Monad.IO.Class (liftIO)
import Control.Monad.Trans.Except
import Data.Either (rights, lefts)
import Data.SpreadSheet (SpreadSheet)
import Data.String.Utils (replace)
import Language.Haskell.Interpreter hiding (eval)
import qualified Language.Haskell.Interpreter as I (eval)
-- internal imports
import Data.Cell
import Lib.Cell
import Lib.Indexing
import Lib.Parsing

-- | 'context' configures the modules and the environment of the cell
-- evaluation. The loaded modules are:
--
-- - Prelude
-- - ExternalModule
-- - Data.SpreadSheet
-- - Data.SpreadSheet.Cell
-- - Data.Function
context :: InterpreterT IO ()
```

```haskell
context = do
  loadModules ["ExternalModule.hs"]
  setImports [ "Prelude"
             , "ExternalModule"
             , "Data.SpreadSheet.Date"
             , "Data.SpreadSheet"
             , "Data.SpreadSheet.Cell"
             , "Data.Function"
             , "Data.Either" ]

-- | 'eval' typechecks and evals the content of a cell.
--
-- >>> runExceptT $ eval "1 + 2"
-- Right "3"
-- >>> runExceptT $ eval "1 + True"
-- Left "Won't compile"
eval :: String -> ExceptT Error IO String
eval ""    = return ""
eval input = ExceptT $ do
  typeRes <- liftIO $ runInterpreter $ do { context; typeChecks input }
  evalRes <- liftIO $ runInterpreter $ do { context; I.eval input }
  return $ case (typeRes, evalRes) of
    (Right False, _)      -> Left $ "eval error;\n\t - expression: " <> input
    (Right True, Left _)  -> Left $ "not showable;\n\t - expression: " <> input
    (Right True, Right x) -> Right x
    _                     -> Left $ "unknown error;\n\t expression: " <> input

-- | 'solveDependencies' returns the string with the cell dependencies
-- solved. The @[Index]@ input must be in topological order.
solveDependencies :: SpreadSheet Cell -> [Index] -> String -> Either Error String
solveDependencies _ _ "" = Right ""
solveDependencies _ [] xs = Right xs
solveDependencies state is _ = do
  let is' = reverse is          -- the last element is the string we want to eval
      vs  = fmap (cellToIndexAndVal . flip getCell state) is'
  case lefts vs of
    []  -> Right $ applyValues . rights $ vs
    err -> Left  $ foldr (<>) [] err

-- | 'applyValues'
applyValues :: [(String, String)] -> String
applyValues [] = []             -- this case should never happen
applyValues (x:[]) = snd x
applyValues (x:xs) = applyValues $ (fmap . fmap) (applyValue x) xs
```

A Código fuente

```haskell
-- | 'applyValue'
applyValue :: (String, String) -> String -> String
applyValue (from, to) = replace from to

-- | 'cellToIndexAndVal'
cellToIndexAndVal :: Cell -> Either Error (String, String)
cellToIndexAndVal cell = do
  c <- case content cell of
         Nothing -> Left $ "Error empty cell " <> index
         Just "" -> Left $ "Error empty cell " <> index
         Just x  -> Right x
  c' <- desugarContent c
  return (index, "(" <> c' <> ")")
  where index = intToCol (col cell) <> intToRow (row cell)
```

Fichero `ExternalModule.hs`

```haskell
-- |This module contains a library of functions to deal with the
-- @ExternalModule@
module Lib.ExternalModule where

-- external imports
import Control.Monad.Trans.Except
import Data.Cell
import Data.ExternalModule
import Language.Haskell.Interpreter as I
import Data.Either.Combinators (mapLeft)

-- |'saveAndLoadExternalModule' first saves the content of the
-- @ExternalModule@ using 'saveExternalModuleFile', then checks if
-- compiles loading it on a GHC interpreter session.
saveAndLoadExternalModule :: ExternalModule -> ExceptT Error IO ()
saveAndLoadExternalModule (ExternalModule input) = ExceptT $ do
  liftIO $ saveExternalModuleFile input
  res <- liftIO $ runInterpreter $ I.loadModules ["ExternalModule.hs"]
  return . mapLeft (const "Won't compile") $ res

-- |'saveExternalModuleFile' writes a file with the content of the
-- @ExternalModule@. This file is called @ExternalModule.hs@ and is
-- located in the path where the program is executed.
saveExternalModuleFile :: String -> IO ()
saveExternalModuleFile input = do
```

```
  writeFile "ExternalModule.hs" input
```

Fichero `Indexing.hs`

```
{-| This module contains functions to deal with indices and references
    in spread sheets.-}
module Lib.Indexing where

-- external imports
import Data.Char (ord, chr)
import Text.Read (readMaybe)
import Text.Regex (matchRegexAll)
import GHC.Natural (intToNatural, naturalToInt)
-- internal imports
import Data.Cell
import Data.Parsing

-- | 'rowColToInt' takes a row and a column and returns an index if
-- succeeds.
rowColToInt :: (String, String) -> Maybe Index
rowColToInt (r, c) = do
  r' <- rowToInt r
  c' <- colToInt c
  return (r', c')

-- | 'rowToInt' takes a @String@ and returns a row index value if
-- succeeds.
rowToInt :: String -> Maybe Row
rowToInt = readMaybe

-- | 'colToInt' takes a String and returns a column index value if
-- succeeds.
colToInt :: String -> Maybe Col -- TODO now it does not work with AA AAA...
colToInt (x:_) = Just $ intToNatural $ ord x - 65 -- unsafe
colToInt _ = Nothing

-- | 'intToRow' takes an index row value and returns a row
-- reference.
intToRow :: Row -> String
intToRow = show

-- | 'intToCol' takes an index column and returns a column reference.
intToCol :: Col -> String       -- TODO fix for greater than 27
```

```
intToCol c = chr (naturalToInt $ c + (65)) : []

-- | 'indexToRef'
indexToRef :: Index -> String
indexToRef (x, y) = intToCol x <> intToRow y

-- | 'getIndex' returns the index of a cell
getIndex :: Cell -> Index
getIndex = (,) <$> row <*> col

-- | 'obtainRowCol' obtains the row and the column from an input
-- reference. It returns @Nothing@ if it can not match the row or the
-- column.
obtainRowCol :: String -> Maybe (String, String)
obtainRowCol xs = do
  r <- matchRegexAll rowRegex xs
  c <- matchRegexAll columnRegex xs
  return (snd' r, snd' c)
  where
    snd' (_, x, _, _) = x
```

Fichero Parsing.hs

```
-- | This module contains functions to deal with parsing of a cell
-- content.
module Lib.Parsing
  ( -- * Reference Functions
    parseReferences
    -- ** Auxiliar Reference Functions
  , matchReferences
    -- ** Desugar Functions
  , desugarContent
  , desugarRange
  , desugarList
    -- ** Auxiliar Desugar Functions
  , desugar
  , genRange
  , rangeToIndex
  , toListString
  , toRangeString
  , showRefs
  ) where
```

```haskell
-- external imports
import Data.String.Utils (replace)
import Text.Regex
import Data.Maybe (catMaybes)
-- internal imports
import Data.Cell
import Data.Parsing
import Lib.Indexing

-- | 'parseReferences' obtains all the indices from the references of
-- an input @String@.
parseReferences :: String -> [Index]
parseReferences = catMaybes . map rowColToInt . catMaybes .
                  map obtainRowCol . matchReferences

-- | 'matchReferences' obtains all the matching references in the
-- input.
matchReferences :: String -> [String]
matchReferences = maybe [] f . matchRegexAll referencesRegex
    where f (_, x, xs, _) = x : matchReferences xs

-- | 'desugarContent' parses the input, checks that indices are
-- correct and desugar them and changes the ranges of values. An
-- example:
--
-- >>> desugarContent "(A0:B2)"
-- Right "(A0 A1 A2 B0 B1 B2)"
-- >>> desugarContent "(B2:A0)"
-- Left "Incorrect index"
desugarContent :: String -> Either Error String
desugarContent s = do
  res  <- desugarRange s
  res' <- desugarList res
  return res'

-- | 'desugarRange' is a specialized version of the 'desugar' function
-- to deal with ranges.
--
-- >>> desugarRange "(A1:B2)"
-- Right "(A1 A2 B1 B2)"
-- >>> desugarRange "[A1:B2]"
-- Right "[A1:B2]"
-- >>> desugarRange "(A1:B0)"
-- Left "Incorrect index"
```

A Código fuente

```haskell
desugarRange :: String -> Either Error String
desugarRange = desugar rangeRegex toRangeString

-- | 'desugarList' is a specialized version of the 'desugar' function
-- to deal with lists.
--
-- >>> desugarList "[A0:B3]"
-- Right "[A0,A1,A2,A3,B0,B1,B2,B3]"
-- >>> desugarList "[B3:A0]"
-- Left "Incorrect index"
desugarList :: String -> Either Error String
desugarList = desugar listRegex toListString

-- | 'desugar' is a higher order function that matches a regex with
-- the input string generates the ranges and finally returns the
-- desugarized string.
desugar :: Regex -> (String -> [Index] -> String -> String)
        -> String -> Either Error String
desugar regex f s | res  == Nothing = Right s
                  | otherwise = do
                      let Just (pre, matched, post, _) = res
                      is <- rangeToIndex matched
                      rs <- genRange (fst is) (snd is)
                      post' <- desugar regex f post
                      return $ f pre rs post'
                        where res = matchRegexAll regex s

-- | 'genRange' returns the range between two given indices.
--
-- >>> genRange (0,0) (2,2)
-- Right [(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2)]
-- >>> genRange (2,2) (0,0)
-- Left "Incorrect index"
genRange :: Index -> Index -> Either Error [Index]
genRange (x1, y1) (x2, y2) | x1 <= x2 && y1 <= y2 =
                               Right $ (,) <$> [y1..y2] <*> [x1..x2]
                           | otherwise = Left "Incorrect index"

-- | 'rangeToIndex' translates a range to its indices.
--
-- >>> rangeToIndex "(A1:B2)"
-- Right ((1,0),(2,1))
-- >>> rangeToIndex "(A1:)"
-- Left "Cannot parse index"
```

```
rangeToIndex :: String -> Either Error (Index, Index)
rangeToIndex = toTuple . parseReferences
  where toTuple (x:y:[]) = Right (x, y)
        toTuple _        = Left "Cannot parse index"

-- | 'toListString'
toListString :: String -> [Index] -> String -> String
toListString pre xs post = pre <> "[" <> (replace " " "," $ showRefs xs) <> "]"<> post

-- | 'toRangeString'
toRangeString :: String -> [Index] -> String -> String
toRangeString pre xs post = pre <> showRefs xs <> post

-- | 'showRefs'
showRefs :: [Index] -> String
showRefs = unwords . map indexToRef
```

Fichero `ServerState.hs`

```
-- | This module contains a library of functions to deal with the
-- @ServerState@
module Lib.ServerState where

-- external imports
import Control.Concurrent
-- internal imports
import Data.Cell
import Data.ServerState
import Lib.Cell
import Lib.Dependency

-- | 'updateState' updates the state of the server with the given cell
-- and its dependencies.
updateState :: MVar ServerState -> Cell -> (Index, [Index]) -> IO ()
updateState state cell deps = do
  modifyMVar_ state $ updateCell cell
  modifyMVar_ state $ updateDependencies deps

-- | 'updateDependencies' updates the state of the dependencies of the
-- @ServerState@.
updateDependencies :: Monad m => (Index, [Index]) -> ServerState -> m ServerState
updateDependencies (from, tos) (ss, ds) = return (ss, updateDependency from tos ds)
```

A Código fuente

```haskell
-- | 'updateCell' updates the @ServerState@ adding the new
-- cell, or replaces it if already exists.
updateCell :: Monad m => Cell -> ServerState -> m ServerState
updateCell cell (ss, ds) = return (addCell cell ss, ds)
```

A.1.3 Paquete app

Fichero Main.hs

```haskell
{-| This module contains the main program of the project. -}
module Main where

-- external imports
import Control.Concurrent
import Control.Monad (forever)
import Control.Monad.Trans.Except
import Control.Monad.Trans.Maybe
import Data.Aeson (decode)
import qualified Network.WebSockets as WS
-- internal imports
import Data.Cell
import Data.ExternalModule
import Data.Messages
import Data.ServerState
import Lib.Application
import Lib.ExternalModule

application :: MVar ServerState -> WS.ServerApp
application state pending = do
  putStrLn "[INFO]: Connection started!"
  conn <- WS.acceptRequest pending
  forever $ do
    msg <- WS.receiveData conn
    case (decode msg) :: Maybe Cell of
      Nothing   -> return ()
      Just cell -> do evalCell state conn cell
                      updateDependents state conn cell
    case (decode msg) :: Maybe Save of
      Nothing       -> return ()
      Just (Save x) -> save state x
    case (decode msg) :: Maybe Load of
      Nothing       -> return $ Right ()
      Just (Load x) -> runExceptT $ load state conn x
```

```
    case (decode msg) :: Maybe ExternalModule of
      Nothing     -> return $ Right ()
      Just extMod -> do runExceptT $ saveAndLoadExternalModule extMod
                        runMaybeT  $ updateSpreadSheet state conn
                        return $ Right ()

main :: IO ()
main = do
  state <- newMVar newServerState
  putStr "[INFO]: Starting Server... "
  saveExternalModuleFile "module ExternalModule where\n"
  putStrLn "Done!"
  WS.runServer "127.0.0.1" 9160 $ application state
```

A.2 Interfaz gráfica de usuario

A.2.1 Paquete `Component`

Fichero `Render/Lib.purs`

```
module Render.Lib where

import Prelude

import Component.Data (Query(..), State)
import Data.Array (catMaybes, head, length, singleton, tail, zipWith, (..))
import Data.Cell (Cell(..), SpreadSheet)
import Data.Cell.Lib (toRowsArray, id)
import Data.Char (fromCharCode)
import Data.Maybe (maybe)
import Data.String.CodeUnits as S
import Data.Tuple (Tuple(..), fst, snd)
import Halogen as H
import Halogen.HTML as HH
import Halogen.HTML.Events as HE
import Halogen.HTML.Properties as HP

renderError :: forall a b. String -> HH.HTML a b
renderError error = HH.li_ [ HH.text error ]

renderSpreadSheet :: forall a. SpreadSheet Cell -> Array (HH.HTML a (Query Unit))
renderSpreadSheet = map HH.tr_ <<< appendColumnHeaders <<< appendRowsHeaders
```

A Código fuente

```
                    <<< map (map renderCell) <<< toRowsArray
  where appendColumnHeaders xs = [ renderColumnHeaders $ columns xs ] <> xs
        appendRowsHeaders   xs = zipWith (\x y -> append [x] y) (map renderRowHeader
                                                          $ 0 .. (length xs)) xs
        columns = maybe 0 (length) <<< head

renderColumnHeaders :: forall a. Int -> Array (HH.HTML a (Query Unit))
renderColumnHeaders cols = append [HH.th_ [] ] <<< map renderColumnHeader <<< catMaybes
                          <<< map fromCharCode $ (65 .. (63 + cols)) -- TODO

renderColumnHeader :: forall a. Char -> HH.HTML a (Query Unit)
renderColumnHeader x = HH.th_ [ HH.text $ S.singleton x ]

renderRowHeader :: forall a. Int -> HH.HTML a (Query Unit)
renderRowHeader x = HH.td_ [ HH.text $ show x ]

renderCell :: forall a. Cell -> HH.HTML a (Query Unit)
renderCell cell@(Cell c) = HH.td_
                  [ HH.input [ HP.type_ HP.InputText
                             , HP.value $ show cell
                             , HE.onValueInput $ HE.input  $ Update (Tuple c.row c.col)
                             , HE.onFocusIn    $ HE.input_ $ UpdateFocus (Tuple c.row c.col)
                             , HE.onFocusOut   $ HE.input_ $ Eval (Tuple c.row c.col)
                             ]
                  ]

render :: State -> H.ComponentHTML Query
render state = HH.div [ HP.class_ (HH.ClassName "container") ]
               [ HH.div
                 [ HP.class_ (HH.ClassName "formula-wrapper") ]
                 [ HH.text $ "fx : " <> fst state.selectedCell ]
               , HH.div
                 [ HP.class_ (HH.ClassName "table-wrapper")]
                 [ HH.table_ [ HH.thead_ $ maybe [] singleton (head spreadSheetHTML),
                               HH.tbody_ $ maybe [] id (tail spreadSheetHTML) ] ]
               , HH.div
                 [ HP.class_ (HH.ClassName "error-wrapper")]
                 [ HH.text $ snd state.selectedCell ]
               , HH.div
                 [ HP.class_ (HH.ClassName "text-input-wrapper") ]
                 [ HH.textarea [ HE.onValueInput $ HE.input  $ UpdateExternalModule
                               , HE.onFocusOut   $ HE.input_ $ SendExternalModule
                               , HP.value $ state.externalModule ] ]
               , HH.div
```

```
                [ HP.class_ (HH.ClassName "save-load-wrapper") ]
                [ HH.div
                  [ HP.class_ (HH.ClassName "path-wrapper") ]
                  [ HH.textarea [ HE.onValueInput $ HE.input  $ UpdateFilePath
                                , HP.placeholder $ "/Users/..."] ]
                , HH.div
                  [ HP.class_ (HH.ClassName "buttons-wrapper") ]
                  [ HH.button [ HE.onClick (HE.input_ SaveFile) ] [HH.text "save"]
                  , HH.button [ HE.onClick (HE.input_ LoadFile) ] [HH.text "load"]
                  ] ] ]
  where spreadSheetHTML = renderSpreadSheet state.spreadSheet
```

Fichero `Communication.purs`

```
module Component.Communication where

import Prelude

import Component.Data (Message(..), Query(..))
import Control.Coroutine as CR
import Control.Coroutine.Aff (emit)
import Control.Coroutine.Aff as CRA
import Control.Monad.Except (runExcept)
import Data.Argonaut (decodeJson, jsonParser)
import Data.Either (either)
import Data.ExternalModule (ExternalModule(..))
import Data.Foldable (for_)
import Data.Maybe (Maybe(..))
import Effect.Aff (Aff)
import Effect.Class (liftEffect)
import Effect.Console (log)
import Foreign (F, Foreign, readString, unsafeToForeign)
import Halogen as H
import Web.Event.EventTarget as EET
import Web.Socket.Event.EventTypes as WSET
import Web.Socket.Event.MessageEvent as ME
import Web.Socket.WebSocket as WS

-- A consumer coroutine that takes output messages from our component
-- IO and sends them using the websocket
wsSender :: WS.WebSocket -> CR.Consumer Message Aff Unit
wsSender socket = CR.consumer \msg -> do
  case msg of
```

```
    OutputMessage msgContents ->
      liftEffect $ WS.sendString socket msgContents
  pure Nothing

-- A producer coroutine that emits messages that arrive from the
-- websocket.
wsProducer :: WS.WebSocket -> CR.Producer String Aff Unit
wsProducer socket = CRA.produce \emitter -> do
  listener <- EET.eventListener \ev -> do
    for_ (ME.fromEvent ev) \msgEvent ->
      for_ (readHelper readString (ME.data_ msgEvent)) \msg ->
        emit emitter msg
  EET.addEventListener
    WSET.onMessage
    listener
    false
    (WS.toEventTarget socket)
  where
    readHelper :: forall a b. (Foreign -> F a) -> b -> Maybe a
    readHelper read =
      either (const Nothing) Just <<< runExcept <<< read <<< unsafeToForeign

-- A consumer coroutine that takes the `query` function from our
-- component IO record and sends `AddMessage` queries in when it
-- receives inputs from the producer.
wsConsumer :: (Query ~> Aff) -> CR.Consumer String Aff Unit
wsConsumer query = CR.consumer \msg -> do
  _ <- H.liftEffect $ log $ "RECEIVED: " <> msg
  let json = jsonParser msg >>= \ss -> decodeJson ss
  either
    (H.liftEffect <<< log)
    (query <<< H.action <<< UpdateResult)
    json
  let json = jsonParser msg >>= \ss -> decodeJson ss
  either
    (H.liftEffect <<< log)
    (query <<< H.action <<< UpdateExternalModule <<< (\(ExternalModule t) -> t.text))
    json
  pure Nothing
```

Fichero `Data.purs`

```
module Component.Data where

import Data.Cell (Cell, Col, Row, SpreadSheet)
import Data.Cell.Lib (createSpreadSheet, emptyCell)
import Data.Tuple (Tuple(..))

data Query a = Update (Tuple Row Col) String a
             | UpdateFocus (Tuple Row Col) a
             | Eval (Tuple Row Col) a
             | UpdateResult Cell a
             | UpdateExternalModule String a
             | SendExternalModule a
             | UpdateFilePath String a
             | SaveFile a
             | LoadFile a

data Message = OutputMessage String

type State = { spreadSheet    :: SpreadSheet Cell
             , selectedCell   :: Tuple String String
             , externalModule :: String
             , filePath       :: String }

initialState :: Int -> Int -> State
initialState r c =
  { spreadSheet: createSpreadSheet emptyCell r c
  , selectedCell: Tuple "" ""
  , externalModule: "module ExternalModule where"
  , filePath: "" }
```

Fichero `SpreadSheet.purs`

```
module Component.SpreadSheet where

import Prelude

import Component.Data (Message(..), Query(..), State, initialState)
import Data.Argonaut (encodeJson, stringify)
import Data.Cell (Cell(..))
import Data.Cell.Lib (clearEvalCell, getCell, getContent,
                      getEvalResult, id, showEval, updateCellContent, updateCell)
```

A Código fuente

```
import Data.Either (Either(..))
import Data.ExternalModule (ExternalModule(..))
import Data.Maybe (Maybe(..), maybe)
import Data.Messages (Load(..), Save(..))
import Data.Tuple (Tuple(..))
import Effect.Aff (Aff)
import Halogen as H
import Halogen.HTML as HH
import Render.Lib (render)

component :: H.Component HH.HTML Query Unit Message Aff
component = H.component
            { initialState: const (initialState 100 25)
            , render
            , eval
            , receiver: const Nothing }

eval :: Query ~> H.ComponentDSL State Query Message Aff
eval (Update (Tuple r c) msg next) = do -- updates a cell content on the user's input
  H.modify_ \state -> state { selectedCell = Tuple msg "" }
  H.modify_ \state -> state { spreadSheet = updateCellContent r c msg state.spreadSheet }
  pure next
eval (UpdateFocus (Tuple r c) next) = do -- updates a cell content on the user's focus in
  s <- H.get
  let cell = getCell r c s.spreadSheet
      fx   = maybe mempty getContent cell
      res  = maybe (Right "") (getEvalResult) cell
  H.modify_ \state -> state { selectedCell = Tuple (maybe "" id fx) (showEval res)
                            , spreadSheet = clearEvalCell r c state.spreadSheet }
  pure next
eval (Eval (Tuple r c) next) = do
  s <- H.get
  let cell = getCell r c s.spreadSheet
      json = maybe "" (stringify <<< encodeJson) cell
  when (json /= "") $ H.raise $ OutputMessage json
  pure next
eval (UpdateResult cell@(Cell c) next) = do -- updates the cell content after eval
  H.modify_ \state -> state { spreadSheet = updateCell cell state.spreadSheet }
  pure next
eval (UpdateExternalModule text next) = do
  H.modify_ \state -> state { externalModule = text }
  pure next
eval (UpdateFilePath path next) = do
  H.modify_ \state -> state { filePath = path }
```

```
  pure next
eval (SendExternalModule next) = do
  s <- H.get
  let json = stringify <<< encodeJson $ ExternalModule { text: s.externalModule }
  H.raise $ OutputMessage json
  pure next
eval (SaveFile next) = do
  s <- H.get
  let json = stringify <<< encodeJson $ Save s.filePath
  H.raise $ OutputMessage json
  pure next
eval (LoadFile next) = do
  s <- H.get
  let json = stringify <<< encodeJson $ Load s.filePath
  H.raise $ OutputMessage json
  pure next
```

A.2.2 Paquete `Data.purs`

Fichero `/Cell/Lib.purs`

```
module Data.Cell.Lib where

import Prelude

import Data.Array (fromFoldable, groupBy, (..), catMaybes, (:))
import Data.Array.NonEmpty (toArray)
import Data.Cell (Cell(..), Col, Row, SpreadSheet, Error, Result)
import Data.Char (fromCharCode)
import Data.Either (Either(..))
import Data.Foldable (class Foldable, foldr)
import Data.Map (Map, empty, filterKeys, insert, lookup, values)
import Data.Maybe (Maybe(..), maybe)
import Data.Set (Set, member)
import Data.String.CodeUnits (singleton)
import Data.Tuple (Tuple(..))

-- Lib functions
createSpreadSheet :: forall a. (Tuple Row Col -> a) -> Row -> Col -> SpreadSheet a
createSpreadSheet e rows columns = foldr f empty keys
    where keys = Tuple <$> (0..rows) <*> (0..columns)
          f x = insert x $ e x
```

A Código fuente

```
emptyCell :: Tuple Row Col -> Cell
emptyCell (Tuple r c) = Cell { row: r, col: c, content: mempty, evalResult: Right "" }

updateCellVal :: forall a. Row -> Col -> a -> SpreadSheet a -> SpreadSheet a
updateCellVal r c = insert (Tuple r c)

updateCellContent :: Row -> Col -> String -> SpreadSheet Cell -> SpreadSheet Cell
updateCellContent r c s table = updateCellVal r c cell' table
  where (Cell cell) = maybe (emptyCell $ Tuple r c) id (getCell r c table)
        cell' = Cell $ cell { content = Just s }

toRowsArray :: SpreadSheet Cell -> Array (Array Cell)
toRowsArray = map toArray <<< groupBy (\(Cell x) (Cell y) -> x.row == y.row) <<<
              fromFoldable <<< values

getCell :: forall a. Int -> Int -> SpreadSheet a -> Maybe a
getCell r c = lookup (Tuple r c)

getCells :: forall a. Set (Tuple Row Col) -> SpreadSheet a -> Array a
getCells keys = fromFoldable <<< filterKeys f
  where f k = member k keys

updateCell :: Cell -> SpreadSheet Cell -> SpreadSheet Cell
updateCell (Cell c) = updateCellVal c.row c.col (Cell c)

updateCellEvalResult :: Cell -> SpreadSheet Cell -> SpreadSheet Cell
updateCellEvalResult (Cell c) ss = updateCell (Cell $ cell { evalResult = c.evalResult }) ss
  where (Cell cell) = maybe (emptyCell $ Tuple c.row c.col) id $ getCell c.row c.col ss

updateCells :: SpreadSheet Cell -> Array Cell -> SpreadSheet Cell
updateCells = foldr updateCell

clearEval :: Cell -> Cell
clearEval (Cell c) = Cell $ c { evalResult = Right "" }

clearEvalCell :: Row -> Col -> SpreadSheet Cell -> SpreadSheet Cell
clearEvalCell r c table = maybe table f $ getCell r c table
  where f cell = updateCellVal r c (clearEval cell) table

showError :: Cell -> String
showError (Cell { col: c, row: r, evalResult: (Left res) }) = f res
  where f x = (toColumn c) <> (show r) <> ": " <> x
showError _ = ""
```

```
toColumn :: Int -> String
toColumn = maybe "" (singleton) <<< fromCharCode <<< (+) 65 -- TODO

showErrors :: forall a b. Ord a => Foldable b => Map a Cell -> b a -> Array String
showErrors table = map showError <<< catMaybes <<< foldr f []
  where f x acc = lookup x table : acc

id :: forall a. a -> a
id x = x

getContent :: Cell -> Maybe String
getContent (Cell cell) = cell.content

getEvalResult :: Cell -> Either Error Result
getEvalResult (Cell cell) = cell.evalResult

showEval :: Either Error Result -> String
showEval (Right "")  = ""
showEval (Right res) = "Result: " <> res
showEval (Left  res) = "Error: "  <> res
```

Fichero `Cell.purs`

```
module Data.Cell where

import Prelude

import Control.Alt ((<|>))
import Data.Argonaut (class DecodeJson, class EncodeJson,
                      decodeJson, jsonEmptyObject, (.?), (:=), (~>))
import Data.Either (Either(..), either)
import Data.Map (Map)
import Data.Maybe (Maybe(..))
import Data.Tuple (Tuple)

-- Lib data types
type Row = Int
type Col = Int
type SpreadSheet a = Map (Tuple Row Col) a

type Error  = String
type Result = String
```

```
data Cell = Cell { row :: Row
                 , col :: Col
                 , content :: Maybe String
                 , evalResult :: Either Error Result }

instance showCell :: Show Cell where
  show (Cell { content: Nothing }) = ""
  show (Cell { evalResult: r, content: (Just c) }) = either (const "ERROR") showRight r
    where showRight "" = c
          showRight x  = x

-- JSON data types and instances
instance decodeJsonCell :: DecodeJson Cell where
  decodeJson json = do
    obj <- decodeJson json
    row <- obj .? "row"
    col <- obj .? "col"
    content <- obj .? "content"
    evalCon <- obj .? "evalResult"
    evalObj <- decodeJson evalCon
    evalResult <- (Left  <$> evalObj .? "Left") <|>
                  (Right <$> evalObj .? "Right")
    pure $ Cell { row, col, content, evalResult }

instance encodeJsonCell :: EncodeJson Cell where
  encodeJson (Cell c) = "row"  := c.row ~>
                        "col"  := c.col ~>
                        "content" := c.content ~>
                        "evalResult" := (eitherEncode c.evalResult ~> jsonEmptyObject) ~>
                        jsonEmptyObject
    where eitherEncode = either (\x -> "Left" := x) (\x -> "Right" := x)
```

Fichero `ExternalModule.purs`

```
module Data.ExternalModule where

import Prelude

import Data.Argonaut (class DecodeJson, class EncodeJson, decodeJson,
                      jsonEmptyObject, (.?), (:=), (~>))

data ExternalModule = ExternalModule { text :: String }
```

```
instance encodeJsonExternalModule :: EncodeJson ExternalModule where
  encodeJson (ExternalModule t) = "text" := t.text ~> jsonEmptyObject

instance decodeJsonExternalModule :: DecodeJson ExternalModule where
  decodeJson json = do
    obj  <- decodeJson json
    text <- obj .? "text"
    pure $ ExternalModule { text }
```

Fichero Messages.purs

```
module Data.Messages where

import Data.Argonaut (class EncodeJson, jsonEmptyObject, (:=), (~>))

data Save = Save String
data Load = Load String

instance encodeJsonSave :: EncodeJson Save where
  encodeJson (Save p) = "save" := p ~> jsonEmptyObject

instance encodeJsonLoad :: EncodeJson Load where
  encodeJson (Load p) = "load" := p ~> jsonEmptyObject
```

A.3 Programa principal Main.purs

```
module Main where

import Prelude

import Component.Communication (wsConsumer, wsProducer, wsSender)
import Component.SpreadSheet (component)
import Control.Coroutine as CR
import Effect (Effect)
import Halogen.Aff as HA
import Halogen.VDom.Driver (runUI)
import Web.Socket.WebSocket as WS

main :: Effect Unit
main = do
  connection <- WS.create "ws://127.0.0.1:9160" []
  HA.runHalogenAff do
```

```
    body <- HA.awaitBody
    io   <- runUI component unit body

    -- The wsSender consumer subscribes to all output messages from
    -- our component
    io.subscribe $ wsSender connection

    -- Connecting the consumer to the producer initializes both,
    -- feeding queries back to our component as messages are received.
    CR.runProcess (wsProducer connection CR.$$ wsConsumer io.query)
```

A.4 Ficheros de marcado y estilo

A.4.1 Fichero `app.css`

```
html,body {
    background-color: rgb(236,236,236);
    height: 97vh;
    width: 98vw;
    justify-items: center;
}

body {
    font-family: -apple-system, BlinkMacSystemFont, "Segoe UI",
                 Roboto, Helvetica, Arial, sans-serif, "Apple Color Emoji",
                 "Segoe UI Emoji", "Segoe UI Symbol";
    font-size: 70%;
}

.container {
    display: grid;
    grid-gap: 2px;
    max-height: 100%;
    max-width: 100%;
    grid-template-columns: repeat(3, 1fr);
    grid-template-rows: 30px 1fr 30px;
}

.formula-wrapper {
    grid-column: 1 / 3;
    grid-row: 1;
    font-size: 20px;
```

```css
}

.text-input-wrapper {
    grid-column: 3 / 4;
    grid-row: 2 / 4;
}

.table-wrapper {
    grid-column: 1 / 3;
    grid-row: 2 / 4;
    background-color: #fff;
    overflow: scroll;
}

.error-wrapper {
    grid-column: 1 / 3;
    grid-row: 4;
    font-size: 15px;
}

.save-load-wrapper {
    grid-column: 3;
    grid-row: 4;
    display: grid;
    grid-gap: 10px;
    grid-template-columns: 1fr;
    grid-template-rows: 30px 1fr;
}

.path-wrapper {
    grid-column: 1;
    grid-row: 1;
}

.buttons-wrapper{
    grid-column: 1;
    grid-row: 2;
}

/* table stuff */
table {
    position: relative;
    border: 1px solid #ddd;
    border-collapse: collapse;
```

```
}

td, th {
    white-space: nowrap;
    border: 1px solid #ddd;
    padding: 5px;
    text-align: center;
}

th {
    background-color: #eee;
    position: sticky;
    top: -1px;
    z-index: 2;

    &:first-of-type {
        left: 0;
        z-index: 3;
    }
}

tbody tr td:first-of-type {
    background-color: #eee;
    position: sticky;
    left: -1px;
    text-align: left;
}

input[type=text] {
    border: none;
}

*:focus {
    outline: none;
}

/* textarea properties */
textarea {
    font-family: monospace;
    width: 100%;
    height: 100%;
    font-size: 15px;
    resize: none;
    outline: none;
```

```
    background-color: #fff;
    color: #000;
    border: 2px solid #ddd;
}
```

A.4.2 Fichero index.html

```
<!doctype html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:foo="http://www.w3.org/2000/svg">
  <head>
    <meta charset="utf-8"/>
    <title>FunCell: Functional SpreadSheet</title>
    <link rel="stylesheet" type="text/css" href="app.css">
  </head>
  <body>
    <script src="app.js"></script>
  </body>
</html>
```

A Código fuente

B

Gramática

⟨*exp*⟩ ::= ⟨*infixexp*⟩ ‘::’ [⟨*context*⟩ ‘=>’] ⟨*type*⟩
| ⟨*infixexp*⟩

⟨*infixexp*⟩ ::= ⟨*lexp*⟩ ⟨*qop*⟩ ⟨*infixexp*⟩
| ‘-’ ⟨*infixexp*⟩
| ⟨*lexp*⟩

⟨*lexp*⟩ ::= ‘\’ ⟨*apat1*⟩ ... ⟨*apatn*⟩ ‘->’ ⟨*exp*⟩
| ‘let’ ⟨*decls*⟩ ‘in’ ⟨*exp*⟩
| ‘if’ ⟨*exp*⟩ [‘;’] ‘then’ ⟨*exp*⟩ [‘;’] ‘else’ ⟨*exp*⟩
| ‘case’ ⟨*exp*⟩ ‘of’ ⟨*alts*⟩
| ‘do’ ‘{’ ⟨*stmts*⟩ ‘}’
| ⟨*fexp*⟩

⟨*fexp*⟩ ::= [fexp] ⟨*aexp*⟩

⟨*aexp*⟩ ::= ⟨*qvar*⟩
| ⟨*gcon*⟩
| ⟨*literal*⟩
| ‘(’ ⟨*exp*⟩ ‘)’
| ‘(’ ⟨*exp1*⟩ ‘,’ ... ‘,’ ⟨*expk*⟩ ‘)’
| ‘[’ ⟨*exp1*⟩ ‘,’ ... ‘,’ ⟨*expk*⟩ ‘]’

B Gramática

| ‘[’ ⟨*exp1*⟩ [, ⟨*exp2*⟩] .. [⟨*exp3*⟩]]
| ‘[’ ⟨*exp*⟩ ‘|’ ⟨*qual1*⟩ ‘,’ ... ‘,’ ⟨*qualn*⟩ ‘]’
| ‘(’ ⟨*infixexp*⟩ ⟨*qop*⟩ ‘)’
| ‘(’ ⟨*qop*⟩⟨-⟩ ⟨*infixexp*⟩ ‘)’
| ⟨*qcon*⟩ ‘{’ ⟨*fbind1*⟩ ‘,’ ... ‘,’ ⟨*fbindn*⟩ ‘}’
| ⟨*aexp*⟩⟨*qcon*⟩ ‘{’ ⟨*fbind1*⟩ ‘,’ ... ‘,’ ⟨*fbindn*⟩ ‘}’ ⟨*context*⟩ ::= ⟨*class*⟩
| ‘(’ ⟨*class1*⟩ ‘,’ ... ‘,’ ⟨*classn*⟩ ‘)’

⟨*decls*⟩ ::= ‘{’ ⟨*decl1*⟩ ‘;’ ... ‘;’ ⟨*decln*⟩ ‘}’

⟨*decl*⟩ ::= ⟨*gendecl*⟩

⟨*gendecl*⟩ ::= ⟨*vars*⟩ ‘::’ [⟨*context*⟩ ‘=>’] ⟨*type*⟩
| ⟨*fixity*⟩ [⟨*integer*⟩] ⟨*ops*⟩
|

⟨*ops*⟩ ::= ⟨*op1*⟩ ‘,’ ... ‘,’ ⟨*opn*⟩

⟨*vars*⟩ ::= ⟨*var1*⟩ ‘,’ ... ‘,’ ⟨*varn*⟩

⟨*pat*⟩ ::= ⟨*lpat*⟩ ⟨*qconop*⟩ ⟨*pat*⟩
| ⟨*lpat*⟩

⟨*lpat*⟩ ::= ⟨*apat*⟩
| ‘-’ (⟨*integer*⟩ | ⟨*float*⟩)
| ⟨*gcon*⟩ ⟨*apat1*⟩ ... ⟨*apatk*⟩

⟨*apat*⟩ ::= ⟨*var*⟩ [‘@’ ⟨*apat*⟩]
| ⟨*gcon*⟩
| ⟨*qcon*⟩ ‘{’ ⟨*fpat1*⟩ ‘,’ ... ‘,’ ⟨*fpatk*⟩ ‘}’
| ⟨*literal*⟩
| ‘_’
| ‘(’ ⟨*pat*⟩ ‘)’
| ‘(’ ⟨*pat1*⟩ ‘,’ ... ‘,’ ⟨*patk*⟩ ‘)’
| ‘[’ ⟨*pat1*⟩ ‘,’ ... ‘,’ ⟨*patk*⟩ ‘]’
| ‘~’ ⟨*apat*⟩

⟨*class*⟩ ::= ⟨*qtycls*⟩ ⟨*tyvar*⟩
| ⟨*qtycls*⟩ ‘(’ ⟨*tyvar*⟩ ⟨*atype1*⟩ ... ⟨*atypen*⟩ ‘)’

```
⟨alts⟩    ::= ⟨alt1⟩ ‘;’ ... ‘;’ ⟨altn⟩

⟨stmts⟩   ::= ⟨stmt1⟩ ... ⟨stmtn⟩ ⟨exp⟩ [‘;’]

⟨stmt⟩    ::= ⟨exp⟩ ‘;’
            | ⟨pat⟩ ‘<-’ ⟨exp⟩ ‘;’
            | ‘let’ ⟨decls⟩ ‘;’
            | ‘;’

⟨fbind⟩   ::= ⟨qvar⟩ ‘=’ ⟨exp⟩

⟨type⟩    ::= ⟨btype⟩ [‘->’ ⟨type⟩]

⟨btype⟩   ::= [⟨btype⟩] ⟨atype⟩

⟨atype⟩   ::= ⟨gtycon⟩
            | ⟨tyvar⟩
            | ‘(’ ⟨type1⟩ ‘,’ ... ‘,’ ⟨typek⟩ ‘)’
            | ‘[’ ⟨type⟩ ‘]’
            | ‘(’ ⟨type⟩ ‘)’

⟨var⟩     ::= ⟨varid⟩
            | ‘(’ ⟨varsym⟩ ‘)’

⟨qcon⟩    ::= ⟨qconid⟩
            | ‘(’ ⟨gconsym⟩ ‘)’

⟨qconop⟩  ::= ⟨gconsym⟩
            | ⟨qconid⟩

⟨qop⟩     ::= ⟨qvarop⟩
            | ⟨qconop⟩
```

Este documento esta firmado por

	Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
	Fecha/Hora	Tue Jul 02 22:24:48 CEST 2019
	Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
	Numero de Serie	630
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)