

# Tesis de Licenciatura

## **$\lambda$ Page**

*Un bloc de notas para desarrolladores Haskell*

Departamento de Computación  
Facultad de Ciencias Exactas y Naturales  
Universidad de Buenos Aires



### **Alumno**

Fernando Benavides (LU 470/01)

[greenmellon@gmail.com](mailto:greenmellon@gmail.com)

### **Directores**

Dr. Diego Garbervetsky

Dr. Daniel Gorín

### **Abstract**

El presente documento describe una herramienta para desarrolladores *Haskell* que pretende facilitar la tarea de “debuggear”, analizar y entender código, llamada  **$\lambda$ Page**. Con ella el usuario puede manipular “páginas” de texto libre que contengan expresiones *Haskell*, intentar interpretar estas expresiones independientemente y analizar los resultados obtenidos.

# Índice

<b>1. Estructura del Informe</b>	<b>4</b>
<b>2. Introducción</b>	<b>5</b>
2.1. Motivación . . . . .	5
2.2. Trabajos Relacionados . . . . .	6
2.3. <i>λPage</i> . . . . .	7
<b>3. Descubriendo <i>λPage</i></b>	<b>9</b>
3.1. Instalación . . . . .	9
3.2. Caso de Uso: Aprobando PLP con <i>λPage</i> . . . . .	10
3.2.1. Pasos previos . . . . .	10
3.2.2. Definición de Tipos y Currificación . . . . .	12
3.2.3. Listas por Comprensión . . . . .	13
3.2.4. Alto Orden y Esquemas de Recursión . . . . .	17
3.2.5. Conclusiones . . . . .	27
3.3. Caso de Uso: Ganando al 4 en Línea con <i>λPage</i> . . . . .	28
3.3.1. Introducción . . . . .	28
3.3.2. Primeros Pasos . . . . .	28
3.3.3. Entendiendo el Proyecto . . . . .	31
3.3.4. Utilizando <i>λPage</i> . . . . .	33
3.3.5. Creando al <i>Jugador Computadora</i> . . . . .	40
3.3.6. Conclusiones . . . . .	48
3.4. Otras Características de <i>λPage</i> . . . . .	49
3.4.1. Listas . . . . .	49
3.4.2. Género . . . . .	52

3.4.3. Carga de Módulos por Nombre . . . . .	56
3.4.4. Configuración del Compilador . . . . .	56
<b>4. Desarrollo - ¿Cómo se hizo <math>\lambda</math>Page?</b>	<b>58</b>
4.1. Arquitectura General . . . . .	58
4.2. Diseño . . . . .	63
4.2.1. Concurrencia . . . . .	63
4.2.2. Bottoms . . . . .	63
4.2.3. Integración . . . . .	66
4.3. Implementación . . . . .	67
4.3.1. eprocess . . . . .	67
4.3.2. Servers . . . . .	69
4.3.3. Módulos de $\lambda$ Page . . . . .	71
4.3.4. UI . . . . .	73
<b>5. Resultados</b>	<b>74</b>
5.1. Objetivos Alcanzados . . . . .	74
5.2. Trabajo a Realizar . . . . .	75
<b>6. Agradecimientos</b>	<b>77</b>

## 1. Estructura del Informe

El presente informe pretende presentar a  **$\lambda$ Page**, una herramienta para facilitar el trabajo de los desarrolladores *Haskell*. El mismo se encuentra dividido en cuatro secciones.

La primera es una sección en la que describiremos los motivos que nos llevaron a desarrollar esta herramienta, hablaremos también de otras herramientas similares y presentaremos  **$\lambda$ Page** de modo general, mostrando principalmente el lugar que pretendemos que ocupe dentro del mundo *Haskell*.

En la siguiente sección intentaremos mostrar, a través de dos tutoriales, cómo utilizar  **$\lambda$ Page** y daremos a conocer sus virtudes y capacidades. Para comenzar, aprenderemos cómo instalarlo, de modo que el lector pueda, una vez instalado el sistema, seguir los tutoriales paso a paso y realizar sus propias experiencias.

Luego, presentaremos un tutorial destinado al público académico que nos mostrará cómo utilizar la herramienta para ayudar al alumno a resolver ejercicios prácticos típicos de varias materias de la facultad. Veremos allí la facilidad de trabajo que brinda  **$\lambda$ Page** al alumno permitiéndole descubrir paso a paso el lenguaje y sus características principales.

El segundo tutorial que presentaremos está apuntado a aquellos desarrolladores que trabajan con proyectos *Haskell* de dimensiones mayores a lo visto en el ámbito académico. La idea de este tutorial es mostrar cómo  **$\lambda$ Page** puede ayudarlos a entender código existente y también a generar y testear nuevo código de manera sencilla y veloz.

Una vez observado  **$\lambda$ Page** en funcionamiento y destacadas sus características principales, observaremos cómo ha sido diseñado y construido. Podremos ver los requerimientos que guiaron su diseño, la arquitectura conceptual que lo subyace y las principales decisiones de diseño e implementación que se han tomado durante su desarrollo.

Finalmente observaremos los resultados obtenidos y los contrastaremos con los objetivos planteados al inicio de este desarrollo. Estableceremos el estado del proyecto en general, cuáles son los siguientes pasos a dar y qué otras tareas pueden llevarse a cabo a partir de ahora.

## 2. Introducción

### 2.1. Motivación

Motivation is what gets you started. Habit is what keeps you going

---

Jim Rohn

Essstamo mo-ti-va-do, nene

---

El “Bambino” Veira

Actualmente estamos presenciando un importante cambio en el desarrollo de sistemas, gracias al éxito de proyectos como [CouchDB](#) [23], [ejabberd](#) [38] y el chat de [Facebook](#) [36], todos ellos desarrollados utilizando lenguajes del paradigma funcional.

Ejemplos de éstos lenguajes de programación, como [Haskell](#) [22] o [Erlang](#) [11], demuestran ser maduros, confiables y se presentan como una alternativa a los lenguajes tradicionales de otros paradigmas. Sin embargo, los desarrolladores que deciden realizar el cambio de paradigma se encuentran con el problema de la escasez de ciertas herramientas que les permitan realizar su trabajo más eficientemente. Por el contrario, éstas herramientas abundan en el desarrollo de proyectos utilizando lenguajes orientados a objetos. En particular, nuestro foco de atención se centra sobre aquellas herramientas que permiten realizar *debugging* y *entendimiento* de código a través de “*micro-testing*”<sup>1</sup>.

Los desarrolladores *Haskell* cuentan actualmente con las siguientes herramientas para realizar esta tarea:

**GHCi** [19] La consola que provee [GHC](#) [21] permite a los desarrolladores evaluar expresiones, verificar su tipo o su clase. Cuenta también con un [mecanismo de debugging](#) [20] integrado que permite realizar la evaluación de expresiones paso a paso. Pese a ser la herramienta más utilizada por los desarrolladores, *GHCi* tiene varias limitaciones. En particular:

- No permite editar más de una expresión a la vez
- No permite intercalar expresiones con definiciones
- Si bien permite utilizar definiciones, éstas se pierden al recargar módulos
- No es sencillo utilizar en una sesión las definiciones y/o expresiones creadas en sesiones anteriores

**Hugs** [18] *Hugs98* es un intérprete pequeño y portátil de Haskell escrito en C, de modo que funciona en casi cualquier máquina. *Hugs98*, que se utiliza mejor como un sistema de desarrollo de programas para Haskell, es extremadamente rápido para la carga de módulos y tiene la ventaja de un intérprete interactivo (en el que uno puede pasar de un módulo a otro para probar diferentes partes de un programa). Sin embargo, al ser un intérprete, ni siquiera se acerca a igualar el rendimiento en tiempo de ejecución de, por ejemplo, programas compilados utilizando *GHC*. Es, sin duda, el mejor sistema para los recién llegados a aprender

---

<sup>1</sup>Entiéndase *micro-testing* como la tarea de realizar tests eventuales para entender o evaluar algún aspecto de un programa

Haskell. Provee muchas librerías y la versión de Windows tiene una interfaz de usuario gráfica llamada *WinHugs*.

**Hat** [7] Una herramienta para realizar seguimiento a nivel de código fuente. A través de la generación de trazas de ejecución, *Hat* ayuda a localizar errores en los programas y es útil para entender su funcionamiento. Sin embargo, por estar basado en la generación de trazas, requiere la compilación y ejecución de un programa para poder utilizarlo y esto no siempre es cómodo para el desarrollador que puede querer simplemente analizar una expresión particular que incluso quizás no compile aún. Además, su mantenimiento activo parece haber cesado hace más de un año y en su página se observa una importante lista de problemas conocidos y características deseadas.

## 2.2. Trabajos Relacionados

---

If I have seen further it is only by standing on  
the shoulders of giants

Isaac Newton

I like work; it fascinates me. I can sit and look  
at it for hours

---

Jerome Klapka

En el mundo de la programación orientada a objetos podemos encontrar herramientas como [Java Scrapbook Pages](#) [10] para Java [14] y [Workspace](#) [25,27] para SmallTalk [24]. Utilizando estos aplicativos, los desarrolladores pueden introducir pequeñas porciones de código, ejecutarlas y luego inspeccionar y analizar los resultados obtenidos. Un concepto compartido por ambas herramientas es el de presentar “páginas” de texto en las que varias expresiones pueden intercalarse con partes de texto libre y permitir al desarrollador intentar evaluar sólo una porción de todo lo escrito. Estas páginas pueden ser guardadas y luego recuperadas de modo de poder analizar nuevamente las mismas expresiones. Además permiten crear objetos (lo que para los lenguajes funcionales equivaldría a definir expresiones) locales a la página en uso y utilizarlos en ella.

Dentro del paradigma funcional, con un enfoque similar, aunque un poco más orientado a la presentación y visualización de documentos, [Keith Hanna](#) de la Universidad de Kent, ha desarrollado [Vital](#) [15]. Vital es una implementación de un entorno de visualización de documentos para Haskell. Pretende presentar Haskell de una manera apropiada para usuarios finales en áreas de aplicación como la ingeniería, las matemáticas o las finanzas. Dentro de esta herramienta, los módulos Haskell son presentados como documentos en los que pueden visualizarse los valores que en ellos se definen directamente en el lugar en el que aparecen, ya sea de modo textual o gráfico (como “vistas”).

Durante el desarrollo de  [\$\lambda\$ Page](#) hemos tenido que enfrentar varios desafíos relacionados principalmente con el desarrollo de interfaces visuales dentro del paradigma funcional. Volcando el conocimiento adquirido durante ese proceso, hemos desarrollado [wxhNotepad](#) [4] que es, ante todo, una prueba de concepto sobre cómo desarrollar editores de texto con *wxHaskell*. Gracias a [Jeremy O'Donoghue](#), *wxhNotepad* está siendo publicado como [un tutorial](#) [32] en sucesivos artículos en su blog

### 2.3. $\lambda$ **P**age

Ancorché lo ingegno umano faccia invenzioni varie, rispondendo con vari strumenti a un medesimo fine, mai esso troverà invenzione più bella, né più facile né più breve della natura, perché nelle sue invenzioni nulla manca e nulla è superfluo

---

Leonardo da Vinci

La programación intensiva y el uso prolongado de Tetris sólo lleva a ver estructuras de orden y secuencias en la verdulería y a querer apilar los autos para formar líneas sólidas

---

Darío Ruellan

**$\lambda$ P**age [3] se presenta como una herramienta similar al Workspace de *Smalltalk*, que permite a los desarrolladores trabajar con documentos de texto libre que incluyan expresiones y definiciones.  **$\lambda$ P**age es capaz de identificar las expresiones y definiciones válidas y permite al desarrollador inspeccionarlas, evaluarlas, conocer su tipo y, en el caso de expresiones de tipo, conocer su género (o *kind*).

En el espíritu de las herramientas provistas por la comunidad de desarrolladores *Haskell*,  **$\lambda$ P**age se integra con *Cabal* [17] y *Hayoo!* [16] y se encuentra ya disponible en *HackageDB* [35]. *Cabal* (Common Architecture for Building Applications and Libraries) es una API distribuida con *GHC* que permite a un desarrollador agrupar fácilmente un conjunto de módulos para producir un paquete. Es el sistema de compilación estándar para las aplicaciones y librerías de *Haskell*. *Hayoo!* es un motor de búsqueda especializado en la documentación de la API de *Haskell*. El objetivo de *Hayoo!* es proporcionar una interfaz de búsqueda interactiva y fácil de usar para la documentación de varios paquetes y librerías. Por su parte, *HackageDB* es un repositorio en internet de versiones de software desarrollado en *Haskell*, almacenadas en paquetes *Cabal*.

**$\lambda$ P**age presenta una interfaz simple e intuitiva, desarrollada utilizando *wxHaskell* [33], lo que lo convierte en una aplicación multiplataforma.

Por ser una herramienta desarrollada con *Haskell* para *Haskell*,  **$\lambda$ P**age se diferencia de sus pares del mundo de objetos, al aprovechar conceptos claves como son el tipado estático, que permite detectar errores de tipo velozmente, evitando el costo de evaluar expresiones complejas, y la evaluación perezosa, que permite evaluar expresiones infinitas e ir exhibiendo resultados progresivamente.

A diferencia de *GHCi* que es una herramienta “de consola”,  **$\lambda$ P**age permite visualizar resultados de manera más dinámica, permitiendo que errores intermedios, detectados durante la evaluación de una expresión no impidan continuar con la misma hasta llegar a un resultado más completo.

**$\lambda$ Page** se encuentra desarrollado utilizando **eprocess** [2], una librería que facilita el manejo de “threads” en un estilo similar al de los procesos *Erlang*. Utilizando de esta librería,  **$\lambda$ Page** puede realizar tareas en paralelo y por lo tanto permitir al usuario continuar editando los documentos en los que está trabajando mientras espera que se evalúe una expresión e incluso cancelar una evaluación conservando la porción del resultado obtenida hasta ese momento. Por otra parte,  **$\lambda$ Page** utiliza eprocess para detectar cálculos infinitos e informar sobre este hecho al usuario para que ya no siga esperando indefinidamente el resultado de la evaluación solicitada.

### 3. Descubriendo $\lambda$ **P**age

#### 3.1. Instalación

As a rule, software systems do not work well until they have been used, and have failed repeatedly, in real applications.

Dave Parnas

The #1 programmer excuse for legitimately slacking off: “My code is compiling”

David Knutz

Para instalar  $\lambda$ **P**age en OSX o Windows, se proveen instaladores en el sitio web de  $\lambda$ **P**age [3], pero, como ya se ha dicho,  $\lambda$ **P**age se encuentra en *HackageDB* y por lo tanto el modo oficial de instalarlo es utilizando *Cabal*, con el siguiente comando:

```
$ cabal install hpage
```

Sin embargo, para ello, previamente se deben satisfacer las siguientes dependencias:

**wxWidgets 2.8.10+** [37] El framework de desarrollo para interfaces de usuario que utiliza *wxHaskell*.

**Haskell Platform** [8] Una distribución de *Haskell* que incluye todo lo necesario para compilar e instalar programas desarrollados en este lenguaje (de particular interés para  $\lambda$ **P**age: *GHC* y *Cabal*).

El proceso de instalación de estas librerías y  $\lambda$ **P**age a través de *Cabal* varía según la plataforma en la que se lo deseé instalar. Las instrucciones detalladas y actualizadas se encuentran disponibles en la *wiki* del sitio web de  $\lambda$ **P**age [6].

## 3.2. Caso de Uso: Aprobando PLP con $\lambda$ **P**age

How is education supposed to make me feel smarter? Besides, every time I learn something new, it pushes some old stuff out of my brain - remember when I took that home winemaking course, and I forgot how to drive?

---

Homer Simpson

Mostraremos a continuación, a través de un ejemplo, cómo utilizar  $\lambda$ **P**age. En este caso, hemos tomado prestada una práctica de la materia *Paradigmas de Lenguajes de Programación* [9]. Exhibiremos entonces, cómo un alumno podría utilizar  $\lambda$ **P**age en el proceso de resolver algunos de los ejercicios que allí se presentan o verificar las soluciones que propone para otros. Seleccionamos sólo aquellos que a nuestro criterio son los más representativos a la hora de entender cómo  $\lambda$ **P**age ayuda al alumno.

### 3.2.1. Pasos previos

Antes de comenzar a resolver los ejercicios, el alumno ejecuta  $\lambda$ **P**age y hace click en el botón New. El programa presentará una pantalla similar a la de la Figura 1.

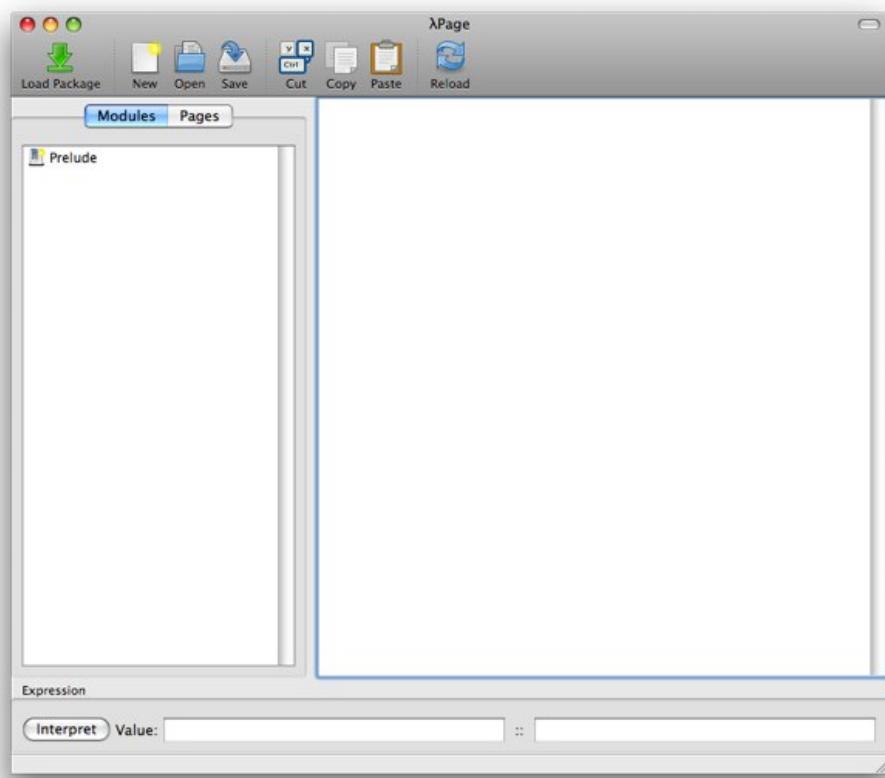


Figura 1: Tutorial 1 - Previo a comenzar

### 3.2.2. Definición de Tipos y Currificación

**Ejercicio 1** Dado el siguiente programa, ¿Cuál es el tipo de ys?

```
xs = [ 1 , 2 , 3 ] :: [ Float ]
ys = map (+) xs
```

**Uso de  $\lambda$ Page** En el caso de este ejercicio, el alumno debería deducir que el tipo de ys es  $[Float \rightarrow Float]$ . Para chequear su deducción y asegurarse de haber obtenido el resultado deseado, puede ingresar el código provisto por la cátedra en la página de  $\lambda$ Page, separando ambas expresiones por un renglón en blanco (pues ese es el modo utilizado por  $\lambda$ Page para determinar qué porción de la página corresponde a cada expresión) y agregando la expresión de la que desea conocer el tipo (ys) al final. Luego de ello, simplemente presiona el botón Interpret y puede observar el tipo del resultado ( $[Float \rightarrow Float]$ ), tal como lo muestra la Figura 2.

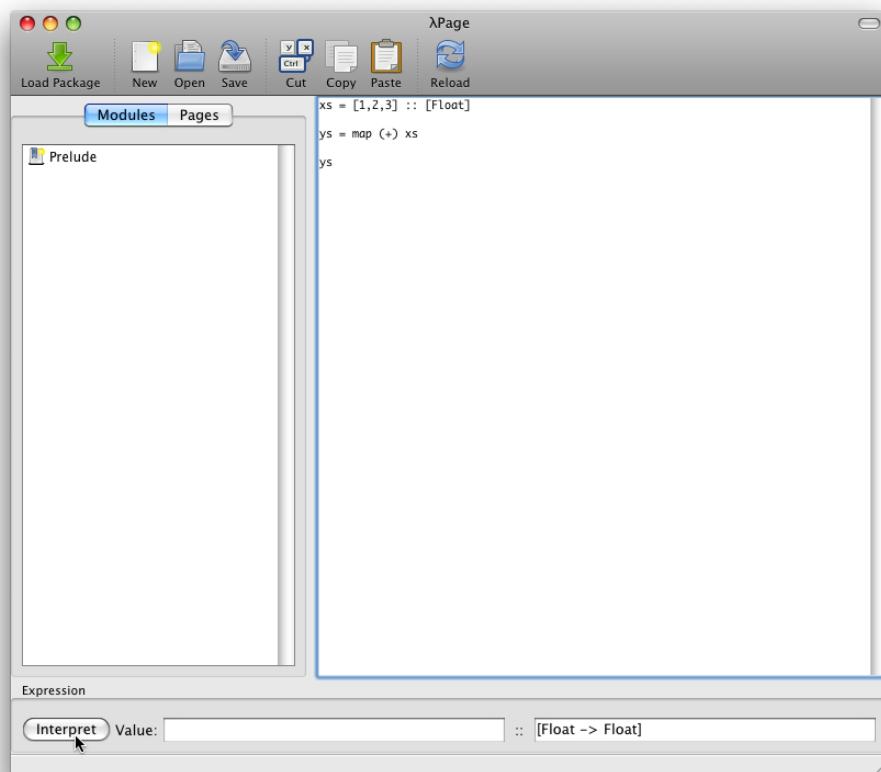


Figura 2: Tutorial 1 - Ejercicio 1

### 3.2.3. Listas por Comprensión

**Ejercicio 4** ¿Cuál es el valor de esta expresión?

```
[ x | x <- [1..4], y <- [x..5], (x+y) `mod` 2 == 0 ]
```

**Uso de  $\lambda$ Page** Nuevamente, en este ejercicio el alumno puede calcular el resultado manualmente y luego utilizar  $\lambda$ Page para chequear su resultado propuesto. Para ello, ingresa el código dentro de la página, selecciona la expresión (pues de ese modo indica a  $\lambda$ Page que, al momento de interpretar, sólo quiere que sea considerada esa porción del texto de la página), y la evalúa presionando el botón Interpret.  $\lambda$ Page mostrará entonces el resultado: [1,1,1,2,2,3,3,4] como se ve en la Figura 3.

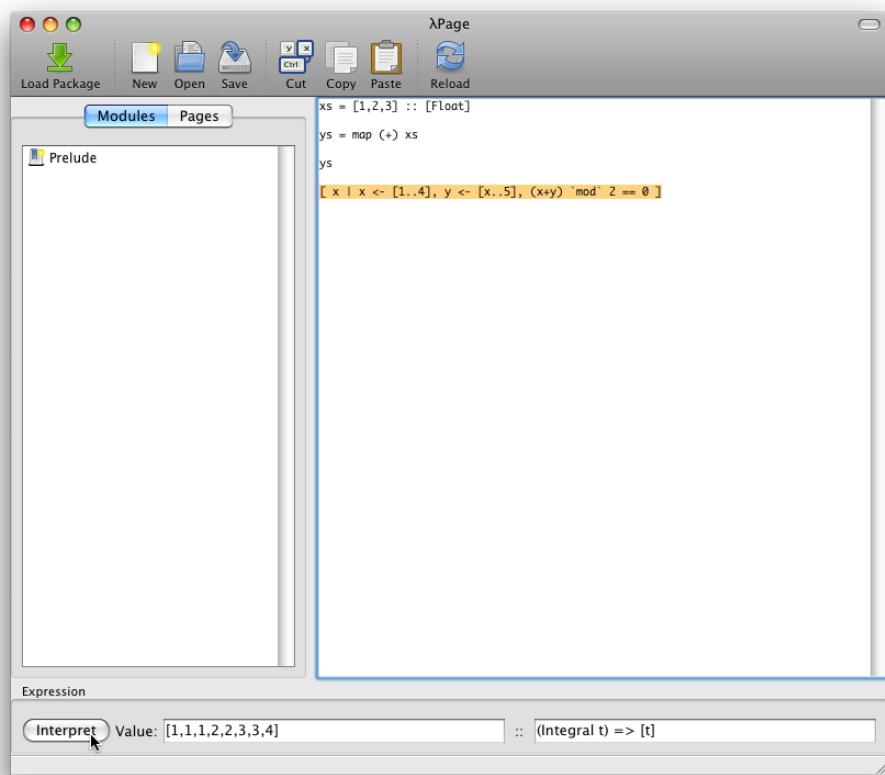


Figura 3: Tutorial 1 - Ejercicio 4

**Ejercicio 5** Una tripla pitagórica es una tripla  $(a, b, c)$  de enteros positivos tal que  $a^2 + b^2 = c^2$ . La siguiente es una definición de una lista (infinita) de triples pitagóricas. Explicar por qué esta definición no es muy útil. Dar una definición mejor.

```

pitagorica :: [(Integer, Integer, Integer)]
pitagorica = [(a, b, c) | a <- [1..], b <- [1..], c <- [1..], a^2 + b^2 == c^2]

```

**Uso de  $\lambda$ Page** Para resolver este ejercicio, el alumno podría comenzar por intentar evaluar la lista que se le provee, para ello escribirá su definición en una página de  $\lambda$ Page tal como se observa en la Figura 4 y, seleccionando las últimas tres expresiones, presionará el botón Interpret. El espacio intermedio tiene la finalidad de distinguir ambas expresiones (por un lado, la declaración de tipo y, por otro, la definición de la expresión)

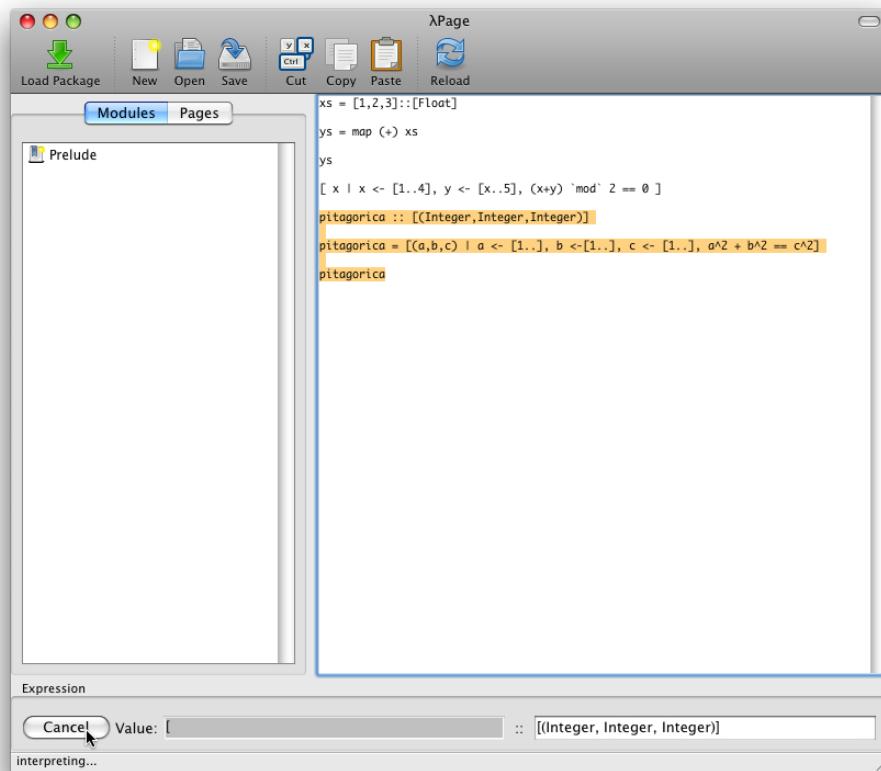


Figura 4: Tutorial 1 - Ejercicio 5 - Primer intento

El alumno podrá observar entonces que el resultado de la interpretación (si bien tiene un tipo válido) nunca aparece por pantalla. Esto se debe al modo en el que se evalúan las listas por comprensión en Haskell: En este caso, teniendo tres generadores (**a**, **b** y **c**) y siguiendo la semántica de Haskell para generar el primer elemento de la lista, el interprete toma el primer valor posible para **a** (o sea 1), el primer valor posible para **b** (o sea 2) y luego itera sobre **c**, con lo que intentará verificar en cada paso de esta iteración que  $1^2 + 1^2 = c$ . Pero  $1^2 + 1^2 = 2$  y sabemos que no existe ningún número natural que elevado al cuadrado sea 2, por lo tanto, el interprete nunca

encontrará el primer elemento de esta lista.  $\lambda\text{Page}$  permite al alumno, pues, presionar el botón *Cancel* de modo de interrumpir la evaluación y poder continuar trabajando.

Luego de presionar el botón *Cancel*, o incluso durante el lapso en el que  $\lambda\text{Page}$  trata de evaluar la expresión, el alumno puede modificar la definición de `pitagorica` para cumplir con la consigna del ejercicio. Podría, por ejemplo, reformularla como muestra la Figura 5 e intentar interpretarla, considerando que, dentro de los números naturales se cumple que  $a > c \Rightarrow a^2 > c^2$  y  $b > c \Rightarrow b^2 > c^2$ .  $\lambda\text{Page}$  entonces, comenzará a exhibir resultados hasta que el alumno presione el botón *Cancel*.

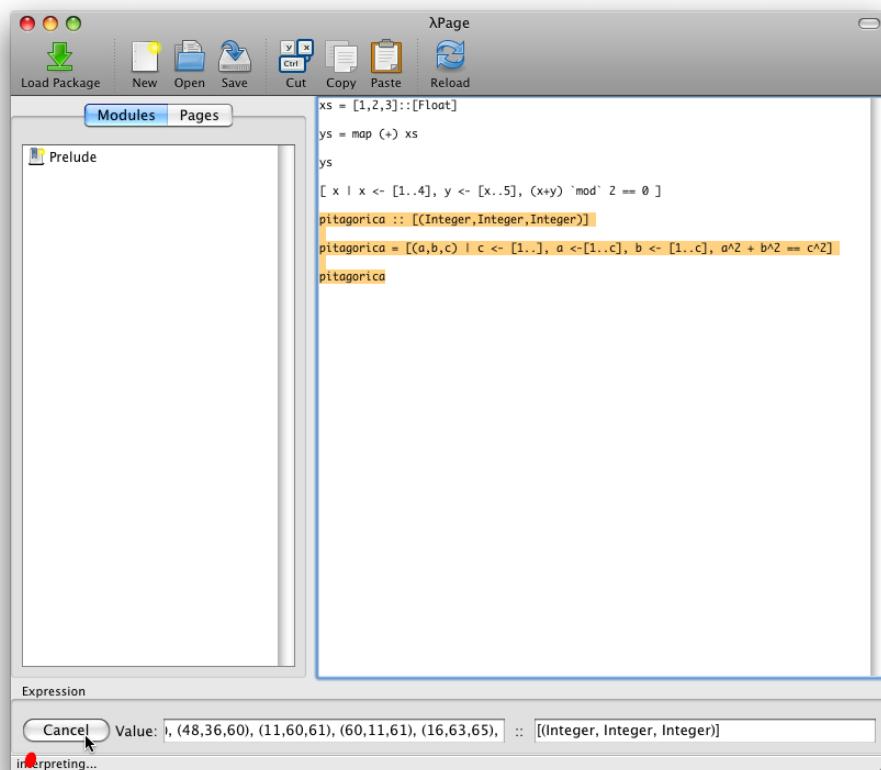


Figura 5: Tutorial 1 - Ejercicio 5 - Segundo intento

Finalmente, el alumno podría también verificar que puede obtener sólo las 5 primeras tuplas pitagóricas, definiendo la serie pitagórica y tomando sólo sus primeros 5 elementos como lo muestra la Figura 6. De este modo no necesitaría presionar el botón *Cancel*.

The screenshot shows the XPage interface with the following details:

- Toolbar:** Includes buttons for Load Package, New, Open, Save, Cut, Copy, Paste, and Reload.
- Left Sidebar:** Shows a "Prelude" module.
- Code Editor:** Displays the following Haskell code:

```
xs = [1..3] :: [Float]
ys = map (+) xs
ys
[ x | x <- [1..4], y <- [x..5], (x+y) `mod` 2 == 0 ]
pitagorica :: [(Integer, Integer, Integer)]
pitagorica = [(a,b,c) | c <- [1..], a <- [1..c], b <- [1..c], a^2 + b^2 == c^2]
take 5 pitagorica
```
- Expression Bar:** Contains an "Interpret" button and a text input field with the value: `[(3,4,5), (4,3,5), (6,8,10), (8,6,10), (5,12,13)] :: [(Integer, Integer, Integer)]`.

Figura 6: Tutorial 1 - Ejercicio 5 - Tercer intento

### 3.2.4. Alto Orden y Esquemas de Recursión

#### Ejercicio 9

- I. Definir la función `genLista`, que genera una lista de una cantidad dada de elementos, a partir de un elemento inicial y de una función de incremento entre los elementos de la lista. Dicha función de incremento, dado un elemento de la lista, devuelve el elemento siguiente.
- II. Usando `genLista`, definir la función `dh`, que dado un par de números (el primero menor que el segundo), devuelve una lista de números consecutivos desde el primero hasta el segundo.

**Uso de `λPage`** En este caso, ciertamente `λPage` no puede ayudar al alumno a *crear* las funciones que se le solicitan, pero sí puede ayudarlo a testearlas. Supongamos pues que el alumno crea una nueva página y define en ella las funciones `genLista` y `dh` tal como se ve en la Figura 7. Luego, intenta testear su ejercicio y, tal como se ve en la Figura 8, puede comprobar que sus funciones generan una recursión infinita. Observa entonces que a `genLista` le falta un **caso base** y lo agrega, para luego volver a testear sus funciones como se ve en la Figura 9 y obtener así el resultado esperado. Al igual que ya hemos visto en varios casos anteriores, debe dejar un renglón intermedio en blanco para que `λPage` distinga las dos partes de la definición de la función.

Cabe aclarar que lo hecho en este ejercicio no es (ni pretende tampoco) ser un completo test de las funciones creadas. Es simplemente lo que hemos llamado *micro-testing*: El ejercicio de realizar pequeñas pruebas “a mano” utilizando expresiones cuyo resultado de evaluación es previsible.

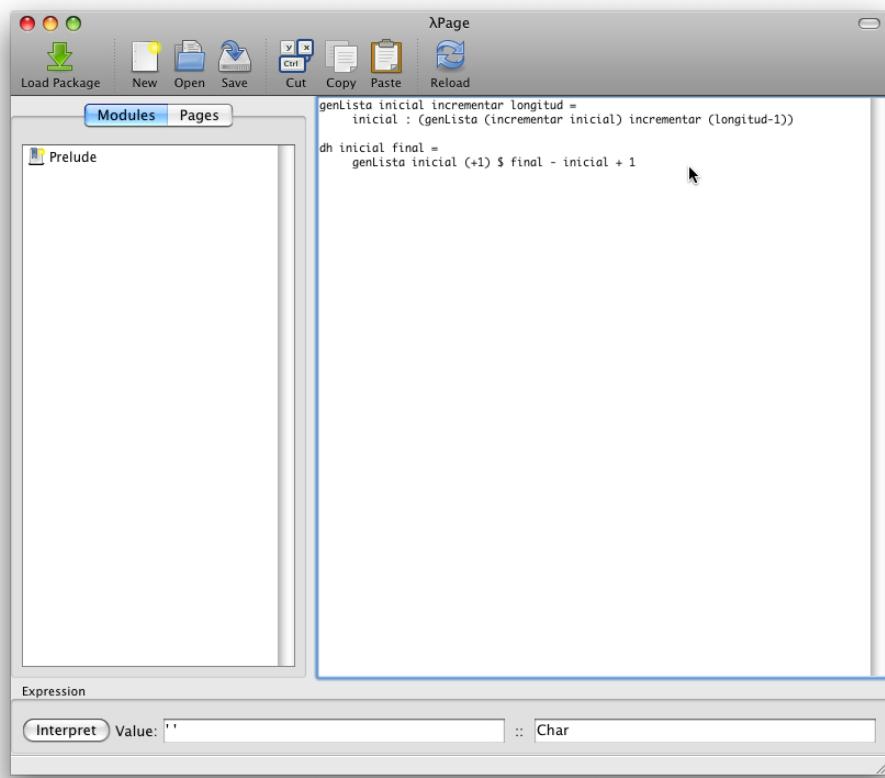


Figura 7: Tutorial 1 - Ejercicio 9 - Primer Intento

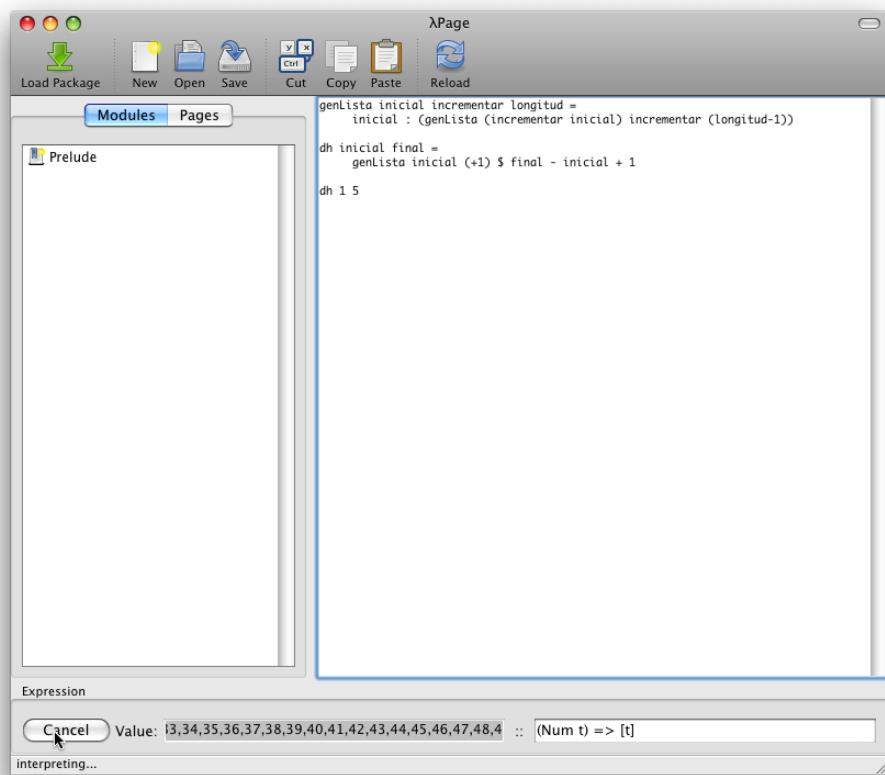


Figura 8: Tutorial 1 - Ejercicio 9 - Recursión Infinita

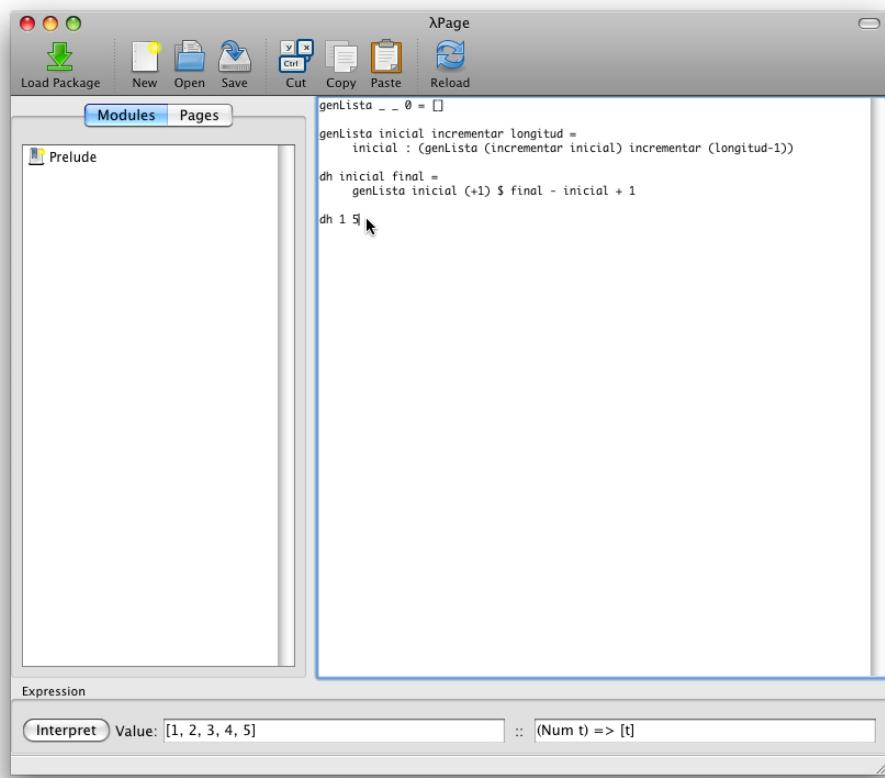


Figura 9: Tutorial 1 - Ejercicio 9 - Segundo Intento

**Ejercicio 23** Definimos el siguiente tipo:

```
data Agenda p t = Vacia | Telefonos p [t] (Agenda p t)
```

Este tipo modela una agenda de teléfonos. A una agenda se le puede agregar una nueva entrada, donde se registra para una persona una lista de teléfonos. Una misma persona puede aparecer en varias entradas. La lista de teléfonos de una entrada puede contener repetidos. Ejemplo:

```
miAgenda = Telefonos "Letincho" [42079999,43834567]
          (Telefonos "Javi" [47779830] (Telefonos "Letincho" [42079999] Vacia))
```

...

**Uso de  $\lambda\text{Page}$**  El ejercicio continúa, pero en este caso, el alumno podría verse tentado a intentar evaluar `miAgenda` directamente en  $\lambda\text{Page}$  y obtendría el resultado de la Figura 10. Ésto se debe a que  $\lambda\text{Page}$  no soporta definiciones de tipos de datos directamente en el texto. Cabe aclarar que el error obtenido no es “prolífico” pues viene directamente de *GHC*, cuya API (utilizada por  $\lambda\text{Page}$  a través de *hint* como se verá en la Sección 4.3) no provee errores tipados, sino solamente descritos en forma de texto.

Para conseguir el efecto deseado, el alumno puede crear un módulo (sin salir de  $\lambda\text{Page}$ ) y guardarlo utilizando la opción *Page → Save As...* o el botón *Save* tal como se observa en la Figura 11. Luego, utilizando la opción *Haskell → Load Modules...* puede cargar el módulo recién creado, seleccionar la expresión `miAgenda` y evaluarla normalmente. Observamos el resultado de esta operación en la Figura 12

Podemos ver, por un lado, que el resultado no ha sido mostrado, sino que sólo se informó su tipo. Esto se debe a que el tipo `Agenda` no es instancia de la clase `Show`. Para visualizar el resultado, el alumno podría agregar la cláusula `deriving (Show)` al tipo `Agenda`, grabar el módulo modificado, presionar el botón *Reload* y luego evaluar nuevamente `miAgenda` tal como se ve en la Figura 13.

Por otra parte, aprovecharemos este ejercicio simple para mostrar lo que  $\lambda\text{Page}$  permite hacer con los elementos de la lista *Modules*. Presionando botón derecho del mouse sobre un ítem,  $\lambda\text{Page}$  despliega un menú contextual que nos permite, entre otras opciones “navegar” el módulo y observar los elementos que exporta. La Figura 14 nos muestra el “árbol” que se desprende de nuestro módulo `Temp`.

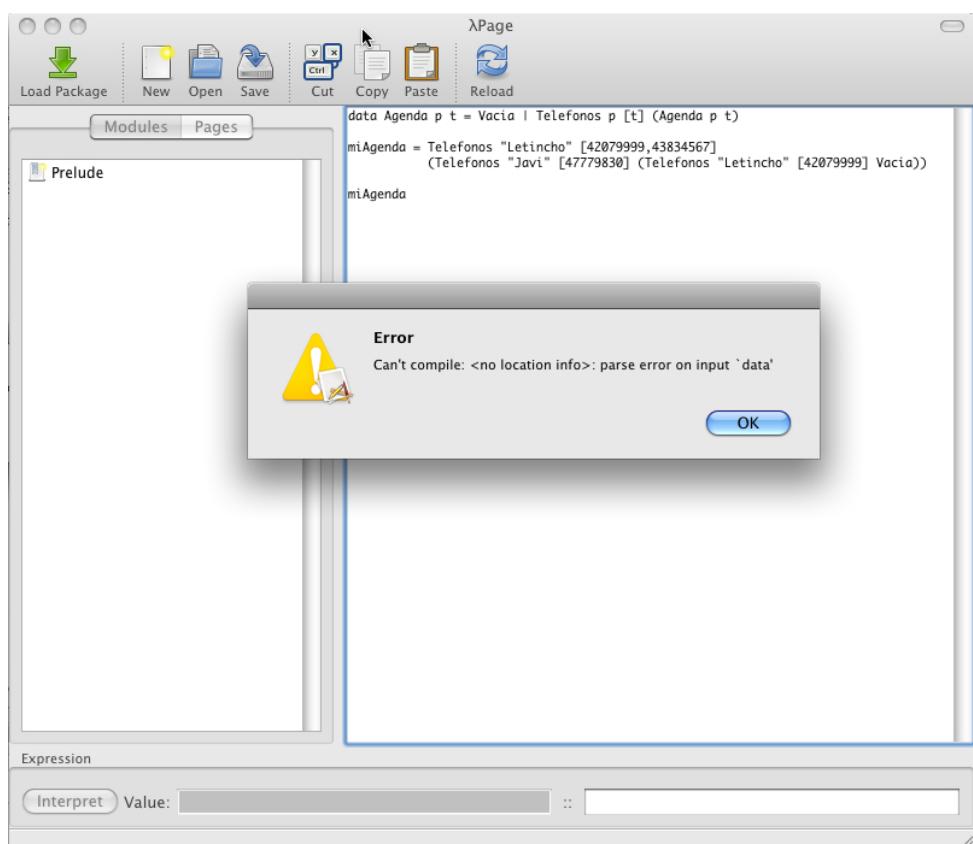


Figura 10: Tutorial 1 - Ejercicio 23 - Primer Intento

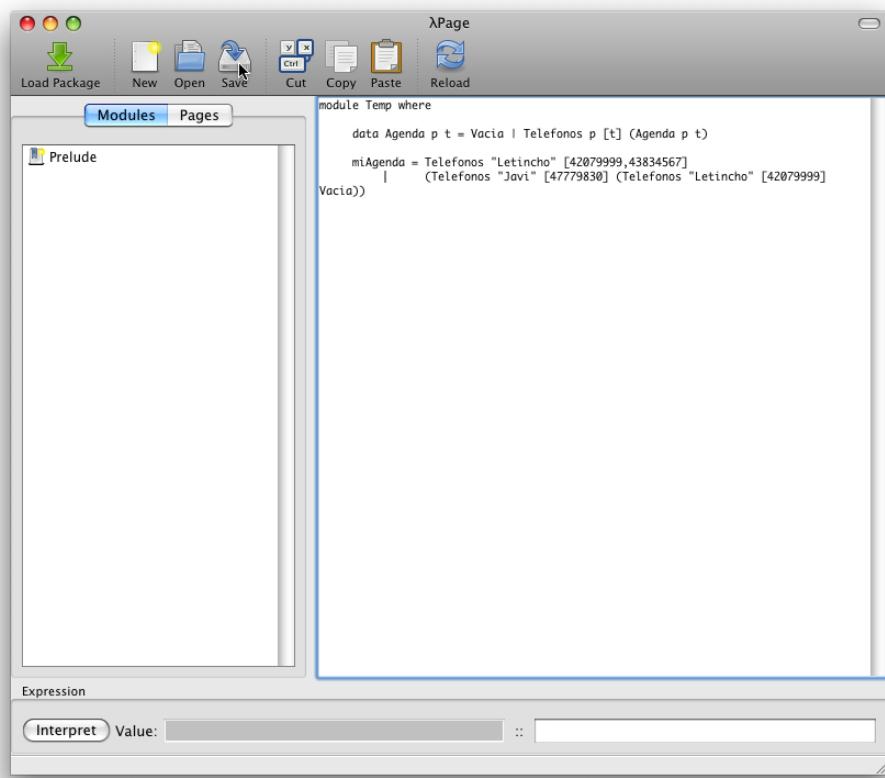


Figura 11: Tutorial 1 - Ejercicio 23 - Crear Módulo

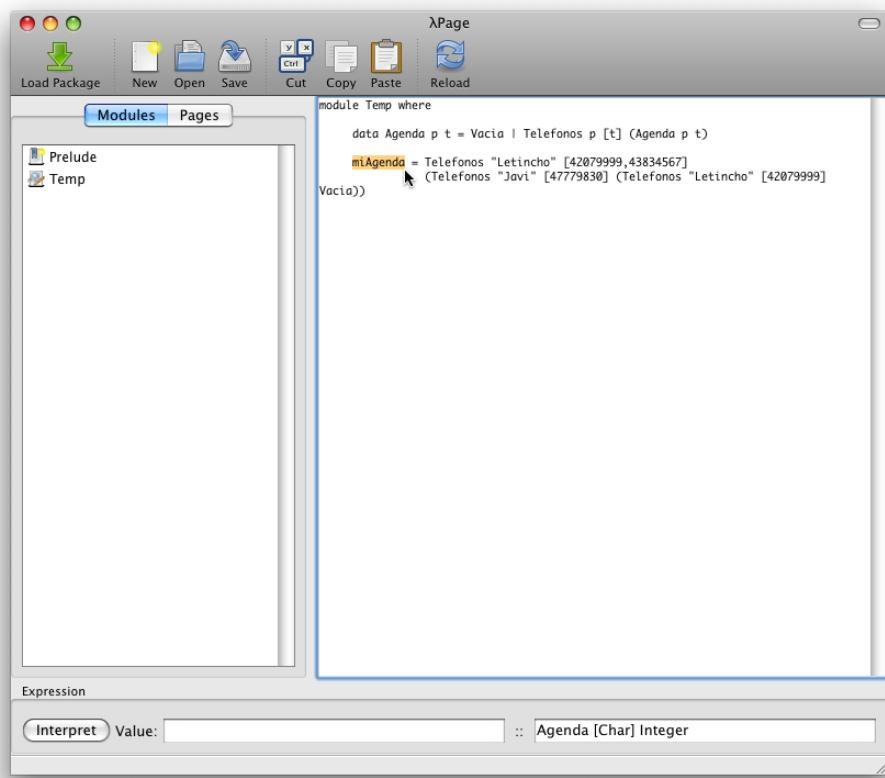


Figura 12: Tutorial 1 - Ejercicio 23 - Segundo Intento

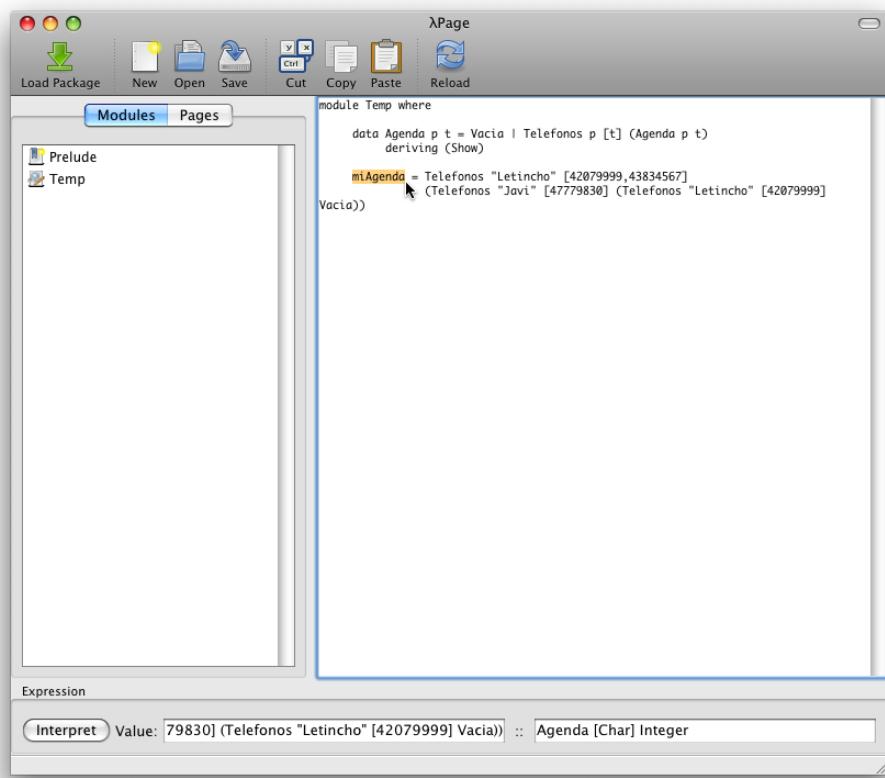


Figura 13: Tutorial 1 - Ejercicio 23 - Tercer Intento

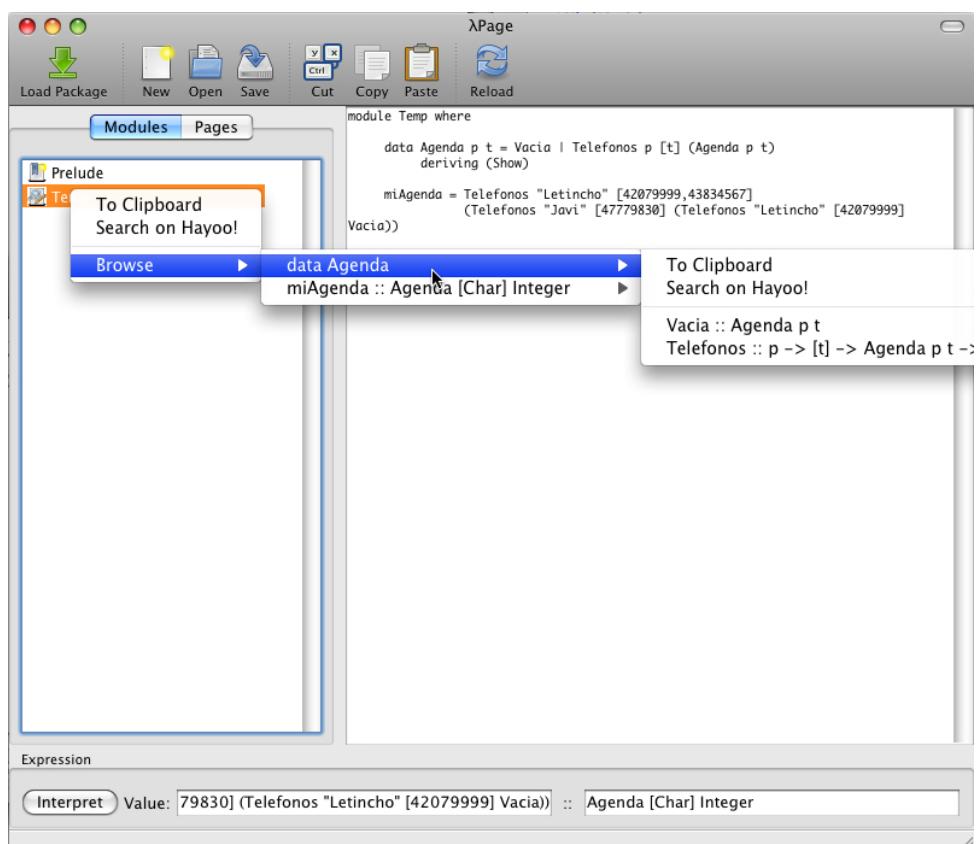


Figura 14: Tutorial 1 - Navegando Módulos

### 3.2.5. Conclusiones

En este tutorial hemos demostrado un uso sencillo de  **$\lambda$ Page** como herramienta de *micro-testing*, permitiendo al usuario trabajar con las funciones definidas en el **Prelude** de *Haskell* además de las que él mismo desee definir localmente.

Pudimos apreciar cómo  **$\lambda$ Page** permite definir expresiones y funciones, para luego evaluarlas de manera individual o combinada. Pudimos ver que  **$\lambda$ Page** distingue unas expresiones de otras por encontrarse separadas utilizando renglones en blanco. Vimos también cómo varía el contexto de evaluación (o sea, las expresiones que  **$\lambda$ Page** toma en consideración al momento de realizar una interpretación):

- Si el usuario seleccionó una porción de texto,  **$\lambda$ Page** sólo toma en consideración las definiciones que se encuentren en la selección e interpreta la última expresión del texto seleccionado
- Si el usuario, en cambio, no ha seleccionado texto alguno,  **$\lambda$ Page** toma en consideración todas las definiciones de la página e interpreta la expresión sobre la que se encuentra posicionado por el cursor

Dichas evaluaciones pueden generar diversos resultados y hemos visto cómo se maneja  **$\lambda$ Page** con algunos de ellos:

- Para aquellas expresiones cuyo valor no puede ser expresado en forma de texto, hemos visto cómo  **$\lambda$ Page** nos permite conocer su tipo.
- En el caso de expresiones cuyo valor es de longitud infinita,  **$\lambda$ Page** exhibe todo lo que el usuario deseé del resultado, culminando cuando éste presiona el botón *Cancel*.
- Y en relación a aquellas que requieren un cálculo infinito para determinar su valor,  **$\lambda$ Page** muestra su tipo y permite al usuario continuar trabajando con las demás expresiones hasta que decida presionar el botón *Cancel* e interrumpir de esa manera el cálculo.

Esta forma de trabajar con los resultados, combinada con la posibilidad que brinda  **$\lambda$ Page** de editar definiciones de manera simple y directa, para luego volver a evaluar expresiones que las usan, permite al usuario trabajar fluidamente y le da la libertad de cometer errores, detectarlos, corregirlos y luego continuar su trabajo. Esta característica de  **$\lambda$ Page** es muy importante sobre todo para quienes se encuentran dando sus primeros pasos en el mundo de *Haskell* pues, entendemos, facilita el aprendizaje del lenguaje.

También hemos visto en este tutorial cómo se puede utilizar  **$\lambda$ Page** para crear, cargar, modificar y recargar módulos, trabajando siempre en una única página de texto. Ésta es otra característica de  **$\lambda$ Page** que facilita el trabajo ya no solamente a los estudiantes sino a todo tipo de desarrolladores *Haskell*.

### 3.3. Caso de Uso: Ganando al 4 en Línea con $\lambda$ Page

We learn by example and by direct experience  
because there are real limits to the adequacy of  
verbal instruction

---

Malcom Gladwell  
Look behind you, a Three-Headed Monkey!

---

Guybrush Threepwood

Incluimos en este informe un segundo tutorial, apuntando esta vez a mostrar cómo  $\lambda$ Page puede ser de utilidad para un programador Haskell que se enfrenta a un problema complejo. En él veremos como  $\lambda$ Page ayuda al usuario a “entender” código escrito por otra persona (o quizá por el mismo, algún tiempo atrás).

#### 3.3.1. Introducción

La historia comienza cuando nuestra amiga desarrolladora, a quien llamaremos Fátima<sup>2</sup> para darle un poco de personalidad, se encuentra con la misión de modificar una implementación de un juego de 4 en Línea, llamada **hfiar** [5]. Fátima tiene que adaptar el juego de modo que permita *jugar contra la computadora* pues actualmente sólo permite jugar a dos seres humanos entre sí.

Como es de suponer, Fátima no conoce al creador del juego y no puede contactarlo por lo que sus únicas herramientas, más allá de su conocimiento de Haskell y del juego en sí, son el código fuente de **hfiar** y  $\lambda$ Page.

#### 3.3.2. Primeros Pasos

Para comenzar, Fátima descarga el código del programa desde *HackageDB* y lo descomprime o bien clona el repositorio *Git* con el siguiente comando:

```
$ git clone git://github.com/elbrujothalcon/hfiar.git
```

Una vez hecho eso, puede observar la estructura del proyecto, tal como se ve en la Figura 15. Conociendo la estructura básica de los proyectos desarrollados en Haskell, podemos describir los archivos allí presentes de la siguiente manera:

**hfiar.cabal** Archivo de descripción de proyecto *Cabal*. Fátima puede obtener de él información general del proyecto, sus módulos, las dependencias y extensiones del lenguaje que se necesitan para compilarlo y demás.

---

<sup>2</sup>El nombre lo hemos elegido en honor a quien, hace ya casi 15 años y utilizando el sobrenombre *Perséfone*, fue la maestra y principal rival de 4 en Línea de Fernando Benavides en *CyberJuegos*

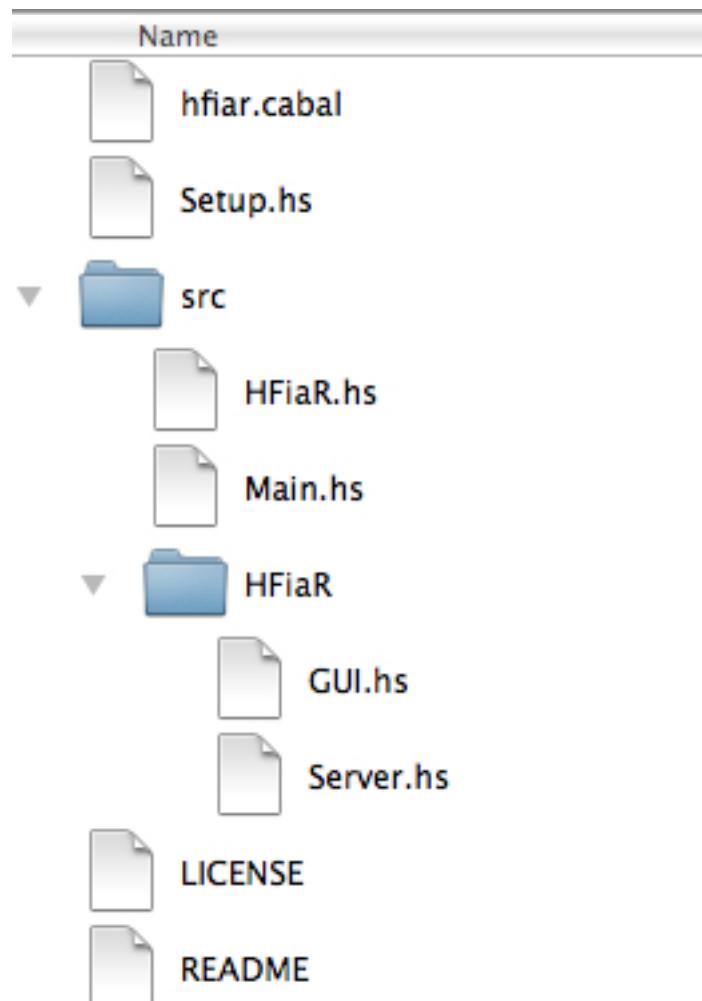


Figura 15: Tutorial 2 - Archivos Originales

**Setup.hs** Archivo Haskell utilizado por *Cabal* para realizar tareas especiales al momento de configurar, compilar o instalar la aplicación. Junto con **hfiar.cabal** permite instalar el proyecto utilizando las siguientes instrucciones:

```
$ cabal configure --user  
$ cabal build  
$ cabal install
```

**src** Carpeta que contiene los archivos de código fuente del proyecto. Fátima deberá analizar cada uno de ellos por separado para poder comprender su funcionamiento.

**LICENSE** Archivo con la licencia del proyecto, en este caso **BSD3** [39].

**README** En el caso de este proyecto, no se trata de algo demasiado útil, dice simplemente:

```
Four in a Row in Haskell!!!  
See http://hackage.haskell.org/package/hfiar
```

### 3.3.3. Entendiendo el Proyecto

Para comenzar a entender cómo está estructurada la aplicación, Fátima observa el archivo **hfiar.cabal** (al que podemos ver en la Figura 16) y observa que el proyecto se encuentra compuesto por una librería (que incluye solamente al módulo **HFiaR**) y un ejecutable llamado **hfiar** que, más allá del módulo **Main**, incluye a los módulos **HFiaR.GUI** y **HFiaR.Server**. Fátima puede ver además que, para compilar los módulos del proyecto, debe utilizar las extensiones **MultiParamTypeClasses** y **GeneralizedNewtypeDeriving**.

Habiendo realizado este análisis, Fátima configura el proyecto ejecutando la siguiente instrucción:

```
$ cabal configure --user
```

Sabiendo que existe una librería en el proyecto, Fátima intenta generar su documentación, ejecutando:

```
$ cabal haddock
```

En el caso de *hfiar*, ese comando genera la documentación en formato HTML, siendo su página principal `dist/doc/html/hfiar/index.html`. Fátima encuentra allí una descripción de los componentes del módulo **HFiaR**. Armada con estos datos, se dispone a utilizar  **$\lambda$ Page** para comprender cómo funcionan esos componentes.

```

name: hfiar
version: 2.0.4
cabal-version: >=1.6
build-type: Custom
license: BSD3
license-file: LICENSE
copyright: 2010 Fernando "Brujo" Benavides
maintainer: greenmellon@gmail.com
stability: stable
homepage: http://github.com/elbruojohalcon/hfiar
package-url: http://code.haskell.org/hfiar
bug-reports: http://github.com/elbruojohalcon/hfiar/issues
synopsis: Four in a Row in Haskell!!
description: The classical game, implemented with wxHaskell
category: Game
author: Fernando "Brujo" Benavides
tested-with: GHC ==6.12.1
data-files: LICENSE README
data-dir: ""
extra-source-files: Setup.hs
extra-tmp-files:

source-repository head
  type: git
  location: git://github.com/elbruojohalcon/hfiar.git

Library
  build-depends: base >= 4,                                base < 5,
                 mtl >=1.1.0,                               mtl < 1.2,
                 eprocess >= 1.1.2,                           eprocess < 2
  extensions: MultiParamTypeClasses, GeneralizedNewtypeDeriving
  exposed-modules: HFiaR
  hs-source-dirs: src

Executable hfiar
  build-depends: wxcore >=0.12.1.4,                      wxcore < 0.13,
                 wx >=0.12.1.4,                         wx < 0.13
  extensions: MultiParamTypeClasses, GeneralizedNewtypeDeriving
  main-is: Main.hs
  buildable: True
  hs-source-dirs: src
  other-modules: HFiaR.GUI, HFiaR.Server
  ghc-options: -Wall

```

Figura 16: Tutorial 2 - hfiar.cabal

### 3.3.4. Utilizando $\lambda$ Page

Una vez abierto  $\lambda$ Page, Fátima puede intentar cargar el proyecto utilizando la opción *Haskell → Load Package...* o el botón *Load Package*. Eso abre una ventana en la que Fátima debe seleccionar el archivo **setup-config** generado por Cabal, tal como se ve en la Figura 17.

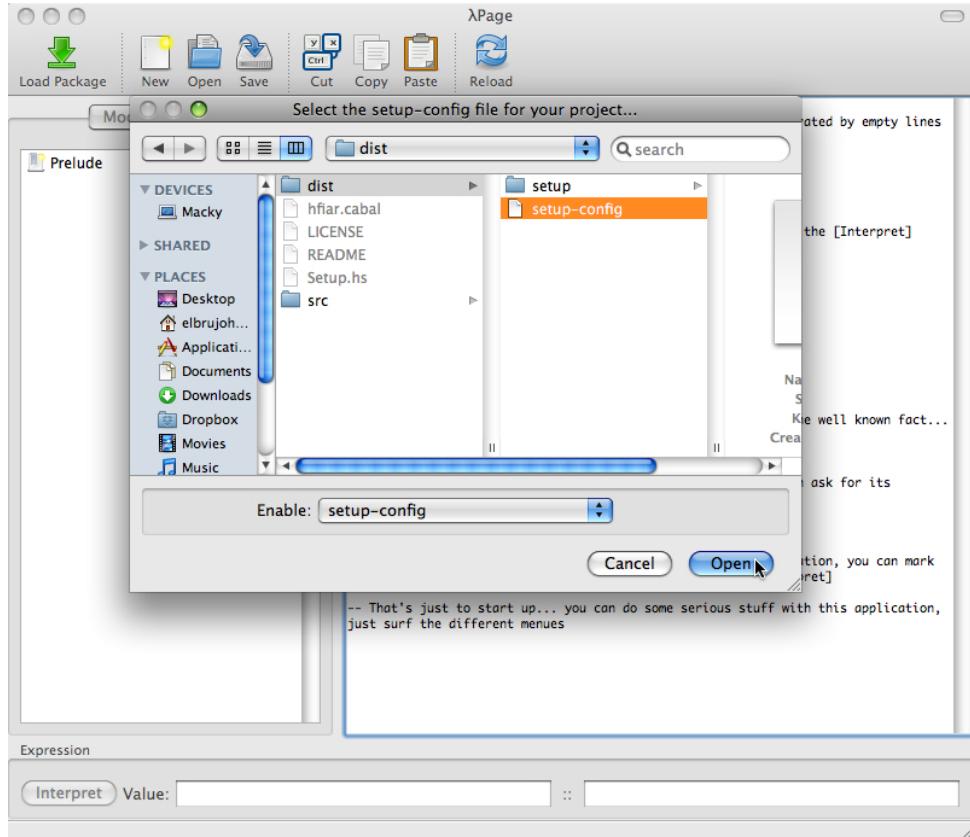


Figura 17: Tutorial 2 - Cargando un proyecto Cabal

Fátima tendrá entonces a su disposición los módulos que componen la aplicación y, haciendo click con el botón derecho del mouse en ellos, podrá cargarlos, como se ve en la Figura 18. Cargar un módulo es la acción de poner en contexto las funciones y tipos de datos definidos en él, de modo que puedan ser utilizados dentro de las páginas actuales. Al cargar un módulo,  $\lambda$ Page carga recursivamente los módulos de los que éste depende. Todos los módulos previamente cargados, excepto los módulos del paquete, se olvidan. Si existe código precompilado para el módulo,  $\lambda$ Page lo utiliza. En otro caso, intenta compilar su código desde el archivo fuente. Cabe destacar que utilizar código precompilado tiene ventajas y desventajas. Por un lado, al estar precompilado, un módulo es cargado más rápidamente y, además, quien lo haya compilado pudo haber utilizado distintas opciones de compilación, por ejemplo, con el objetivo de generar código optimizado. Pero, por otra parte, al tratarse de un módulo precompilado, el usuario de  $\lambda$ Page ya no tendrá acceso a las funciones internas del mismo, como lo tendría si el módulo fuese cargado desde su archivo fuente.

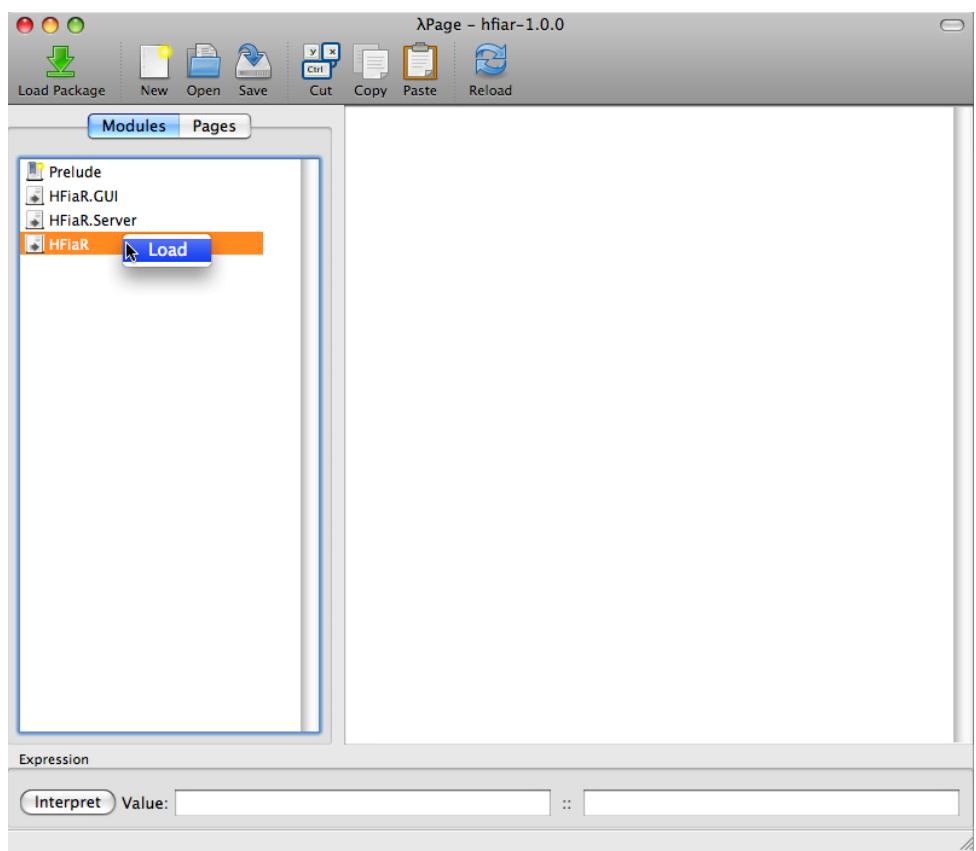


Figura 18: Tutorial 2 - Proyecto *Cabal* cargado

Recordando que el proyecto incluía una librería compuesta únicamente por el módulo `HFiar`, nuestra amiga Fátima decide cargarlo y navegarlo utilizando el menú desplegable que nos muestra la Figura 19.

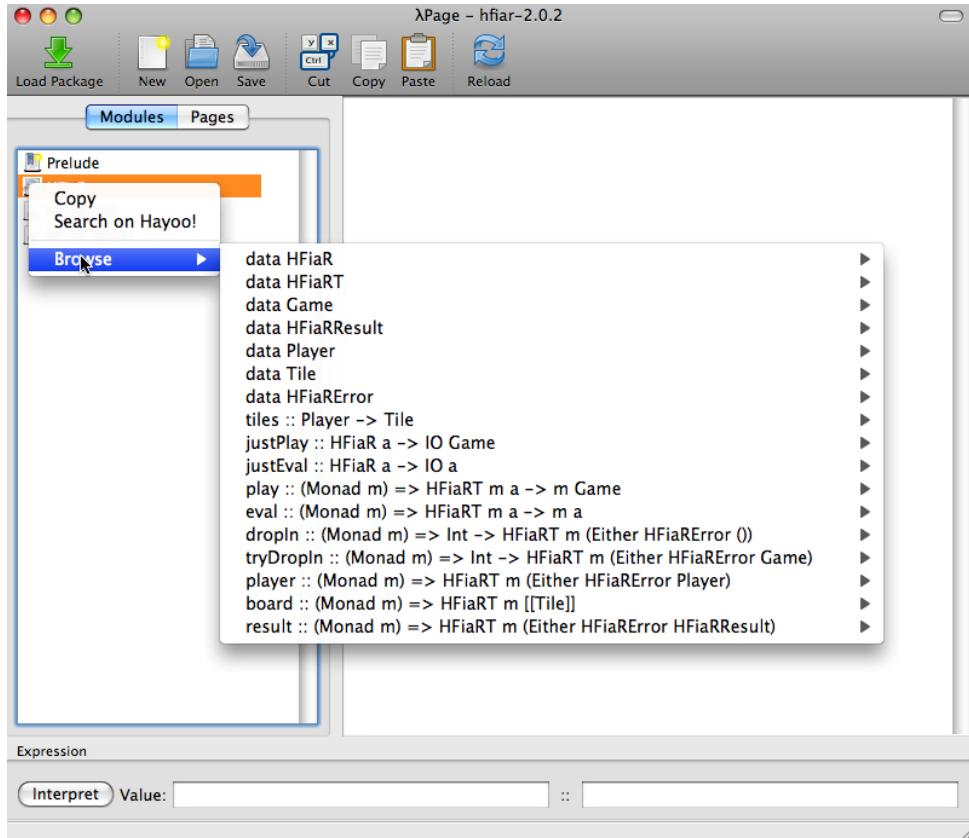


Figura 19: Tutorial 2 - Módulo HFiar

Gracias a la documentación generada previamente, puede observar que la dinámica del juego está modelada con una mónada [40] que describe las acciones llevadas a cabo durante un partido. El concepto de mónada (monad) es uno de los más difíciles de comprender para aquellos que dan sus primeros pasos en *Haskell*. A los fines de este tutorial, podemos ver a las mónadas como tipos de datos que nos permiten definir acciones que generan resultados (por ejemplo, `dropIn` o `player` son acciones de la mónada `HFiar`) y concatenarlas utilizando distintos operadores genéricos (como `>>`) para generar nuevas acciones más complejas (como por ejemplo, `dropIn 1 >> player`, que es una acción de la mónada `HFiar` que representa el hecho de arrojar una ficha en la columna 1 y luego verificar cuál es el jugador actual). Estas acciones, luego pueden ser “ejecutadas”. En nuestro caso dicha ejecución se realiza, por ejemplo, utilizando las funciones `play` (cuyo resultado refleja el estado del partido una vez ejecutada la acción) o `eval` (cuyo resultado es el de la acción ejecutada).

Fátima verifica las funciones de la mónada `HFiar` con las que cuenta:

`dropIn` representa el hecho de arrojar una ficha en una columna

`tryDropIn` representa el hecho de verificar qué pasaría en el caso de realizar serie de jugadas, donde cada una de ellas corresponde a arrojar una ficha en una columna. Es similar a `dropIn` pero sin que el juego avance

`player` le permite conocer el jugador actual

`board` le permite observar el tablero, que es modelado como una lista de listas de `Tiles`, o sea fichas)

`result` le permite verificar el resultado del partido, si es que el mismo ha concluido

Como el lector, al igual que Fátima, habrá podido observar en la Figura 19, si bien las acciones que pueden definirse corresponden a la mónada `HFiaR`, dado el tipo de `play` y `eval`, las mismas pueden ser ejecutadas dentro de cualquier mónada. Esto se debe a que la mónada está implementada utilizando la técnica de *Monad Transformers* [30]. Los *Transformadores de Mónadas* (*Monad Transformers*) son variantes especiales de las mónadas que facilitan la combinación de mónadas. Sus constructores de tipo tienen como parámetro un tipo monádico y producen tipos monádicos combinados. En este tutorial, por ejemplo, las funciones `justPlay` y `justEval`, están definidas internamente utilizando el tipo transformador `HFiaRT` para ejecutar acciones de la mónada `HFiaR` dentro de la mónada `IO`.

Utilizando estas funciones, Fátima comienza a realizar su tarea de *micro-testing*. Su primera prueba consiste en *jugar* un partido en el que nada pasa, sólo para ver qué información puede obtener de él. Coloca entonces en la página la expresión que presentamos a continuación e intenta interpretarla tal como se ve en la Figura 20. Esta expresión, aunque simple, presenta dos elementos muy utilizados por los desarrolladores *Haskell*: el operador `$`, que equivale a encerrar entre paréntesis lo que sigue en la expresión, facilitando la lectura de la expresión en su totalidad, y la expresión `return ()`, que es la acción que representa el hecho de simplemente “no hacer nada”. Como toda acción debe tener un resultado el de `return ()` es `()`, el único valor posible de tipo `Unit`.

```
justPlay $ return ()
```

El resultado obtenido es el que sigue y puede interpretarse como “el partido se encuentra en curso, es el turno del jugador que juega con fichas verdes y el tablero está vacío”. Cabe notar que el tipo del resultado es `IO Game`, o sea, una acción con posibles efectos colaterales de entrada/salida (dentro de la mónada `IO`) cuyo resultado es un juego (`Game`). Sin embargo, el resultado obtenido representa simplemente a un juego. Esto se debe a que `λPage`, siguiendo el ejemplo de *GHCi*, trata de manera especial a las acciones de la mónada `IO` y, en lugar de intentar mostrarlas (cosa que, por cierto, no podría hacer) las evalúa y luego intenta presentar su resultado.

```
OnCourse {gamePlayer = Green ,  
          gameBoard = [ [ ] , [ ] , [ ] , [ ] , [ ] , [ ] , [ ] ] }
```

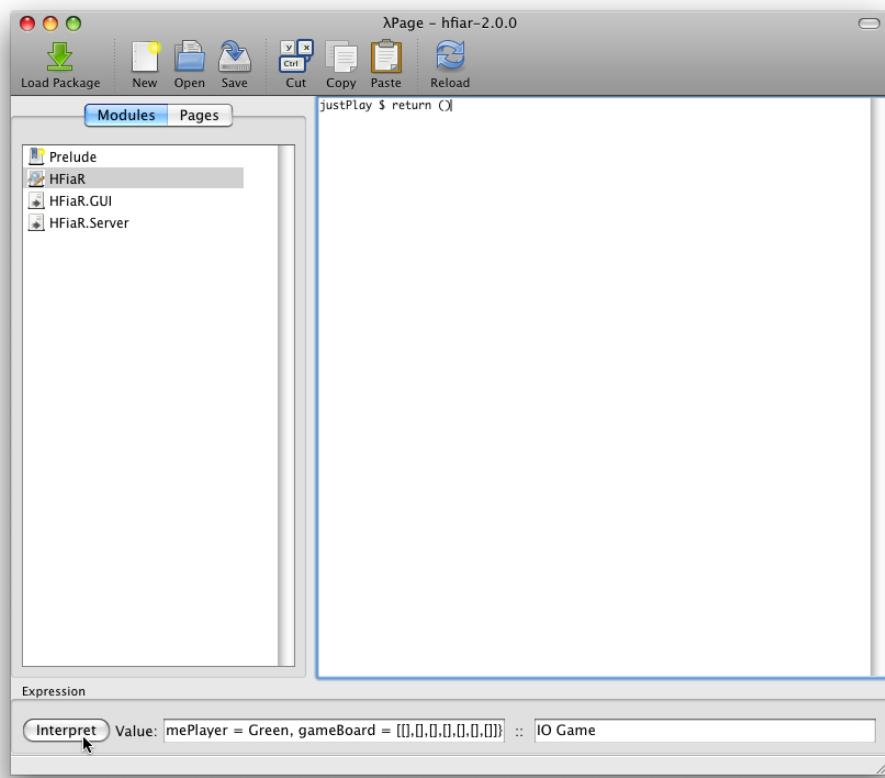


Figura 20: Tutorial 2 - Juego Mínimo

Usando  $\lambda\text{Page}$ , Fátima realiza varias otras pruebas que consignamos en la tabla que presentamos a continuación. Con ellas puede entender el funcionamiento general de la mónada `HFiaR` y de ese modo determinar cómo su jugador de inteligencia artificial puede observar el desarrollo del juego, saber si el mismo ha concluido o no, si es su turno, cuál es el estado del tablero y elegir cómo jugar. Cabe notar que los resultados de tipo `Left HFiaRError` se exhiben como texto porque el desarrollador de `hfiar` ha decidido utilizar la función `show` para describir los errores en lugar de exhibir su constructor.

Expresión	Resultado
justPlay \$ <b>return</b> ()	OnCourse {gamePlayer = Green, gameBoard = [[],[],[],[],[],[],[]]}
justEval player	<b>Right</b> Green
justEval \$ dropIn 1 >> board	[[], [Green], [], [], [], [], []]]
justEval \$ dropIn 1 >> dropIn 2 >> board	[[], [Green], [Red], [], [], [], []]]
justEval \$ dropIn 1 >> dropIn 1 >> dropIn 2 >> board	[[], [Red, Green], [Green], [], [], [], []]]
justEval \$ dropIn 7	<b>Left</b> That column doesn't exist
justPlay \$ dropIn 4 >> dropIn 1 >> dropIn 4 >> dropIn 1 >> dropIn 4 >> dropIn 1 >> dropIn 4	Ended {gameResult = WonBy Green, gameBoard = [[], [Red, Red, Red], [], [], [Green, Green, Green, Green], [], []]}
justEval \$ dropIn 4 >> dropIn 1 >> dropIn 4 >> dropIn 1 >> dropIn 4 >> dropIn 1 >> dropIn 4 >> result	<b>Right</b> (WonBy Green)
justEval \$ dropIn 4 >> dropIn 1 >> dropIn 4 >> dropIn 1 >> dropIn 3 >> dropIn 1 >> dropIn 4 >> result	<b>Left</b> Game is still on course
justEval \$ dropIn 1 >> dropIn 1 >> dropIn 1 >> dropIn 1 >> dropIn 1 >> dropIn 1 >> dropIn 1 >> dropIn 1	<b>Left</b> That column is full

Cuadro 1: Tutorial 2 - Pruebas realizadas en *λPage*

### 3.3.5. Creando al *Jugador Computadora*

Con el conocimiento adquirido en el uso de `HFiaR`, Fátima decide crear la función `aiDropIn`, que se ejecutará dentro de la mónada `HFiaR`. A continuación observamos el tipo de dicha función. Se puede ver que el tipo de la función es similar al de `dropIn`, salvo por el hecho de no recibir el número de columna en la que el jugador desea volcar su ficha. Eso se debe, justamente, a que será la propia función quién lo determine.

```
-- | Drop a tile in a column chosen by the Artificial Intelligence
aiDropIn :: Monad m => HFiaRT m (Either HFiaRError ())
```

Para crear `aiDropIn`, Fátima no cierra  $\lambda \text{Page}$ . Por el contrario, define la función allí mismo, como podemos ver en la Figura 21, componiéndola con otra función a la que denomina `bestColumn`. Esta función se ejecutará sólo en el caso de que el partido no haya concluido. La idea es que en ella estará el algoritmo principal de selección de columna a jugar (o sea, el verdadero motor de *inteligencia artificial*), el cual se irá perfeccionando a lo largo del tutorial. La primera versión de `bestColumn`, con más de artificial que de inteligencia, simplemente elegirá la primer columna que no esté llena y `aiDropIn` arrojará la ficha ahí. Recordamos que, para que  $\lambda \text{Page}$  reconozca las distintas expresiones, Fátima debe separarlas con renglones en blanco.

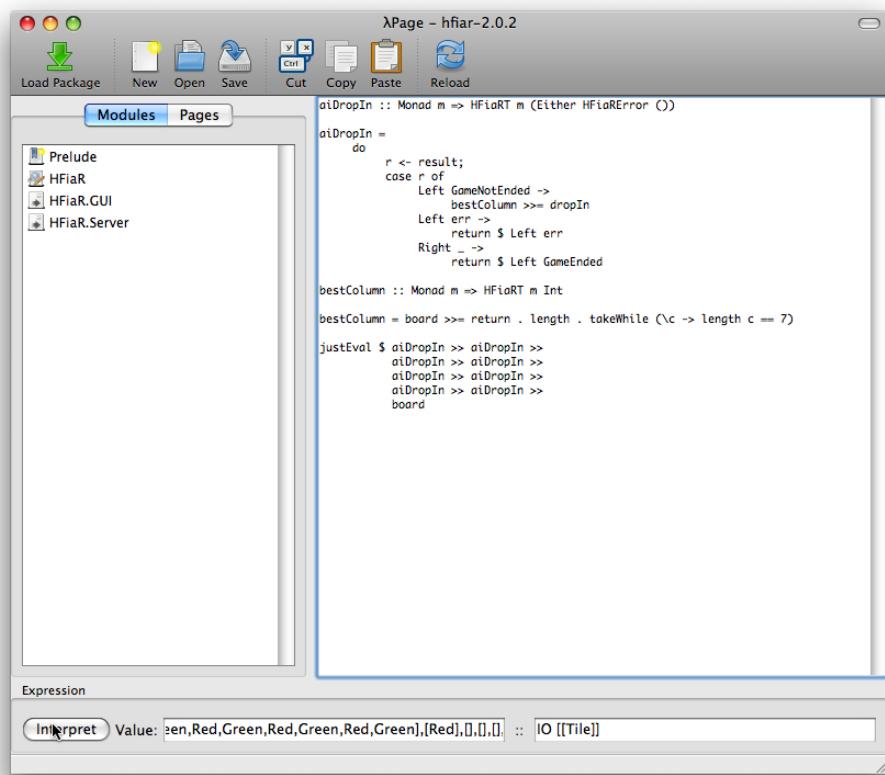


Figura 21: Tutorial 2 - `aiDropIn` versión 1

Una vez definidas ambas funciones, Fátima decide testearlas y, para ello, no necesita más que evaluar un juego en el que conozca el resultado. En particular, Fátima elige hacer jugar a la computadora contra sí misma 8 movimientos y verifica que las fichas han quedado intercaladas en la primer columna y luego el siguiente jugador ha colocado una en la segunda.

El siguiente paso es agregar un poco de inteligencia a `bestColumn` de modo de que realmente sea competitivo. Para ello, Fátima prevé que necesitará varias funciones auxiliares y una buena cantidad de código, por lo que decide convertir su página actual en un módulo al que denomina `HFiaR.AI` y graba en el lugar correspondiente entre los fuentes del proyecto, utilizando la opción *Page → Save As...* o el botón *Save*. A partir de ese momento, Fátima puede utilizar su editor *Haskell* favorito para agregar la cabecera del módulo, la importación del módulo que tenía cargado (`HFiaR`) y “bautizar” la expresión utilizada para testear como `testGame`. Luego lo carga utilizando la opción *Haskell → Load Modules...* y entonces, en una página nueva (creada utilizando la opción *Page → New*) escribe e interpreta la expresión `testGame`, de modo de realizar la misma prueba de la Figura 21. Es interesante notar que, pese a que el módulo creado sólo exporta la función `aiDropIn`, Fátima pudo interpretar sin problemas la función `testGame`, pues había cargado el módulo utilizando el archivo con el código fuente del mismo y eso le da acceso a todas las funciones del mismo, tanto públicas como privadas. A partir de este momento, Fátima utiliza su editor favorito para trabajar en el módulo que ha creado y, ante cada modificación del mismo, lo recarga utilizando la opción *Haskell → Reload* o el botón *Reload*.

Para la segunda (y a los fines de este tutorial, definitiva) versión de `bestColumn`, experta jugadora de 4 en línea como es, Fátima pone un poco de criterio y decide que la función tenga en cuenta las condiciones que enumeramos a continuación:

- Si poniendo la ficha en alguna columna el jugador gana el partido, se debe elegir esa columna.
- Si poniendo la ficha en alguna columna se impide que el jugador rival gane el partido (pues acumula 3 alineadas convenientemente) se debe elegir esa columna.
- En otro caso, de ser posible, debe elegirse la columna 3 (o sea, la columna central) pues para formar líneas de 4 fichas horizontales o diagonales se requiere una ficha en dicha columna

Podemos observar entonces la nueva versión de `bestColumn` y sus funciones adicionales:

```

import HFiaR
import Data.Maybe

bestColumn :: Monad m => HFiaRT m Int
bestColumn =
  do
    j1 <- columnWhereWins
    j2 <- columnWhereLoses
    j3 <- column3IfAvailable
    j4 <- firstAvailableColumn
    return . head $ catMaybes [j1, j2, j3, j4]

columnWhereWins, columnWhereLoses,
  column3IfAvailable, firstAvailableColumn :: Monad m => HFiaRT m (Maybe Int)

columnWhereWins = mapM (tryDropIn . (:[])) [0..6] >>= return . firstEnded

columnWhereLoses = mapM moves [0..6] >>= return . firstEnded
  where moves :: Monad m => Int -> HFiaRT m (Either HFiaRError Game)
        moves col = do
          b <- board
          let avail c = c == col || length (b !! c) == 7
          let other = length $ takeWhile avail [0..6]
          tryDropIn [other, col]

column3IfAvailable = board >>= \b -> return $ case length (b !! 3) of
  7 -> Nothing
  _ -> Just 3

firstAvailableColumn = board >>=
  return . Just . length . takeWhile (\c -> length c == 7)

firstEnded :: [Either HFiaRError Game] -> Maybe Int
firstEnded games = case (length $ takeWhile onCourse games) of
  7 -> Nothing
  g -> Just g
  where onCourse (Left _) = False
        onCourse (Right OnCourse{}) = True
        onCourse (Right Ended{}) = False

```

Muchas porciones del código que acabamos de exponer pueden resultar desconocidas o quizás difíciles de comprender incluso para un programador Haskell ya habituado a este tipo de código. Tomemos, por ejemplo, la función `catMaybes`, utilizada en la nueva versión de `bestColumn`. A los fines de continuar con nuestra historia y demostrar el uso de  $\lambda$ **Page**, supongamos que, si bien Fátima sabía de la existencia de esta función, no sabía en

qué módulo se encontraba declarada ni recordaba exactamente su tipo. En tal caso, lo que Fátima puede hacer es escribir el nombre de la función en la página de  $\lambda$ **Page**, seleccionarlo y utilizar la opción *Search on Hayoo!* del menú contextual que se despliega al presionar el botón derecho del mouse sobre el texto seleccionado. En ese momento,  $\lambda$ **Page** le presentará la página web que se ve en la Figura 22. Allí se puede observar que la función `catMaybes` se encuentra definida en el módulo `Data.Maybe` y que su tipo es `[Maybe a] -> [a]`. Más aún, *Hayoo!* nos muestra la documentación que acompaña a la función, según la cual *la función `catMaybes` toma una lista de `Maybes` y devuelve una lista de todos los valores construidos con `Just`*. Para probarla, Fátima puede importar el módulo `Data.Maybe` utilizando la opción *Haskell → Import modules...* y realizar alguna prueba similar a la de la Figura 23.

El lector podría preguntarse en este momento por qué Fátima debió importar el módulo en lugar de cargarlo como ha hecho con los demás módulos. Esto se debe a que el módulo `Data.Maybe` se encuentra en una librería del sistema, por lo tanto  $\lambda$ **Page** no tiene acceso a su código fuente ni a su versión precompilada. *Importar* un módulo es similar a cargarlo: es la acción de poner en contexto las funciones y tipos de datos definidos y **exportados** por él, de modo que puedan ser utilizados dentro de las páginas actuales. Al importar un módulo, el usuario no tiene acceso a las funciones y tipos de datos privados del mismo.

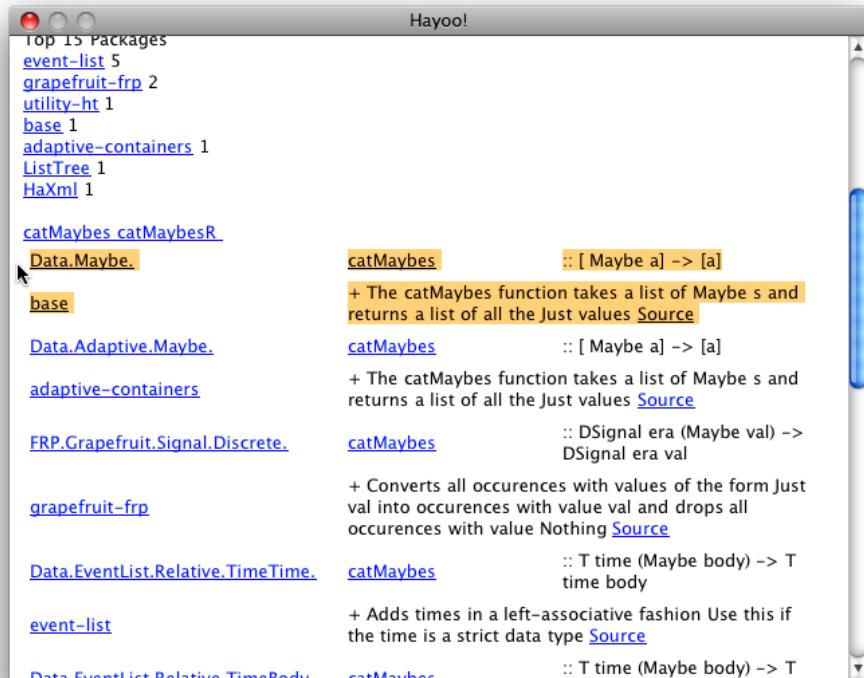


Figura 22: Tutorial 2 - Utilizando *Hayoo!*

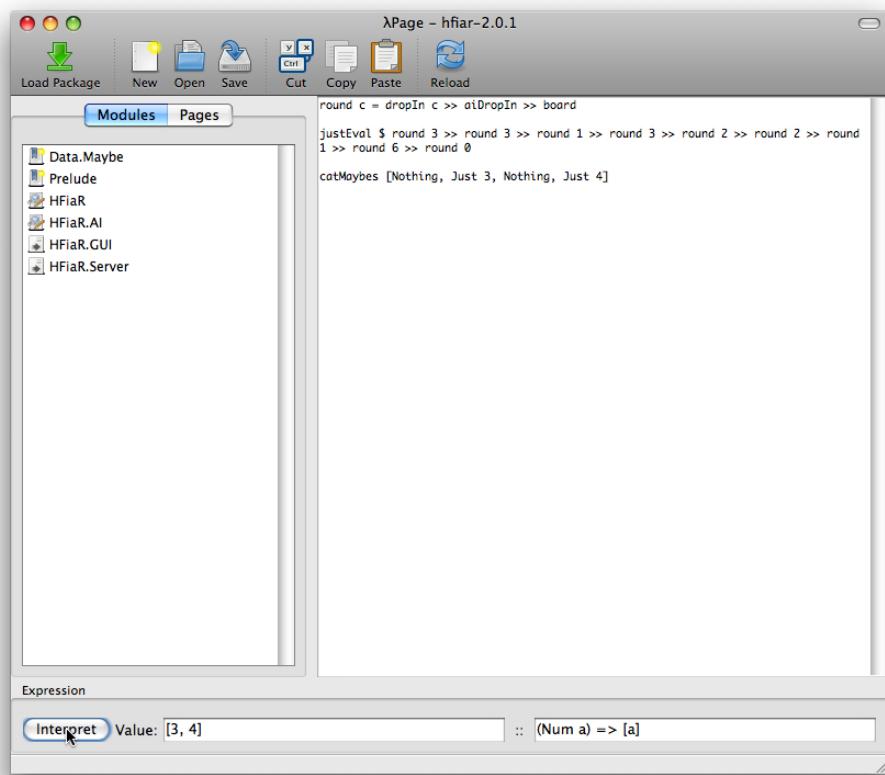


Figura 23: Tutorial 2 - Testeando `catMaybes`

La función `testGame` ya no es suficiente para testear el funcionamiento de `aiDropIn` pero, como el código está en un módulo que ha sido cargado dentro de `λPage`, lo que Fátima puede hacer es ir construyendo paso a paso una expresión que represente un partido entre ella y *el Jugador Computadora*. Para ello, define la función auxiliar `round` y, tal como lo vemos en la Figura 24, la utiliza una y otra vez para modificar el partido que pretende “jugar”. De ese modo puede probar el comportamiento de su módulo de inteligencia artificial ante las distintas circunstancias de un partido tan extenso como ella deseé. Cabe destacar una vez más aquí que, cada vez que Fátima deseé realizar cambios en su módulo, puede hacerlo con su editor favorito, luego presionar el botón `Reload` al volver a `λPage` e interpretar nuevamente sus expresiones, que no han sido perdidas pues están todavía en la(s) página(s) con la(s) que está trabajando.

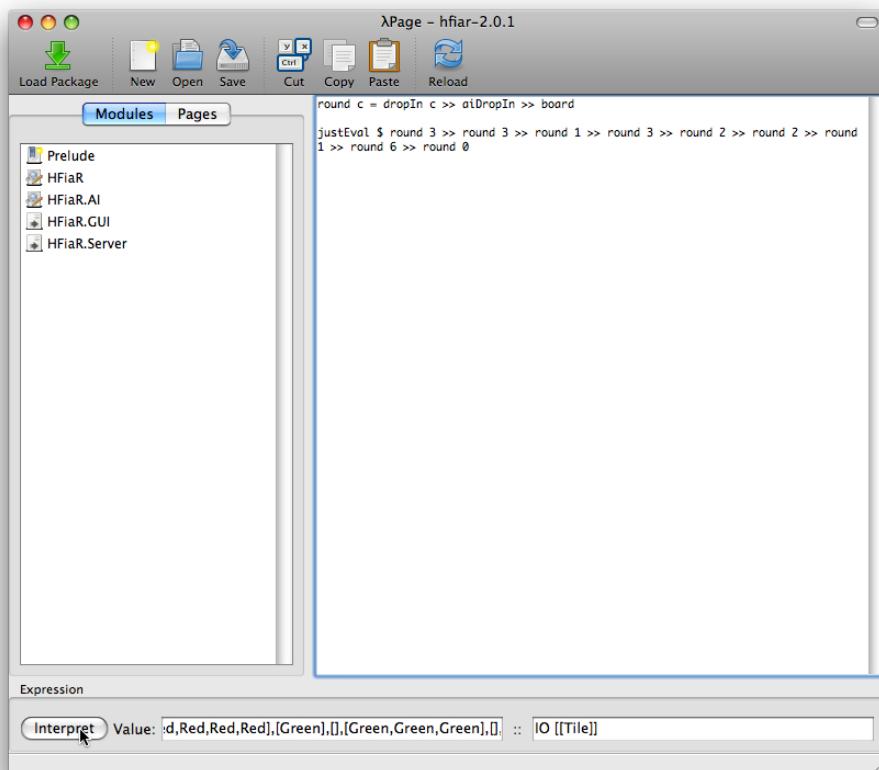


Figura 24: Tutorial 2 - Jugando contra HFiaR.AI

Resta pues modificar la interfaz gráfica para que permita al usuario jugar contra la computadora, utilizando la función `aiDropIn` en los turnos correspondientes a la máquina. Para ello, Fátima abre el módulo `HFiaR.GUI` y realiza los cambios pertinentes. Se trata básicamente de modificaciones de código relacionado vinculado a `wxHaskell`, para las cuales `λPage` no es de más ayuda que en los ejemplos que hemos visto a lo largo de este tutorial. Por ese motivo no los detallamos en este informe. Para quien deseé observarlos puede encontrarlos en la página web del proyecto `hfiar` en [github](#) [5]

El último paso que Fátima debe realizar es incluir su nuevo módulo en la lista de módulos de la librería *HFiaR* dentro del archivo **hfiar.cabal**. Una vez hechas estas modificaciones, puede ejecutar los siguientes comandos y disfrutar de un partido de su juego favorito contra la computadora:

```
$ cabal configure --user  
$ cabal build  
$ cabal install  
$ ~/cabal/bin/hfiar &
```

### 3.3.6. Conclusiones

A lo largo de este tutorial hemos podido ver cómo utilizar  **$\lambda$ Page** para comprender código *Haskell* a través de *micro-testing*. Hemos visto también como, utilizando esa misma técnica, podemos construir paso a paso módulos *Haskell* medianamente complejos. Pudimos ver cómo  **$\lambda$ Page** nos permite, al editar los módulos, recargarlos para volver a testearlos sin perder las expresiones que ya habíamos definido.

Hemos podido ver cómo  **$\lambda$ Page** se encuentra integrado con *Cabal* para permitir cargar proyectos previamente configurados y evitar al desarrollador lidiar con el manejo de extensiones, carpetas y módulos manualmente. Esto es una importante ventaja que ofrece  **$\lambda$ Page** dado que la mayoría de los proyectos desarrollados en *Haskell* se encuentran organizados en paquetes *Cabal*. Pudimos observar también cómo  **$\lambda$ Page** se integra con *Hayoo!*, permitiendo al desarrollador consultar sus dudas sobre la API de *Haskell* directamente dentro de la aplicación.

Hemos observado cómo utilizar  **$\lambda$ Page** para cargar o importar módulos y hemos notado las diferencias entre estas dos acciones. También hemos observado la forma gráfica con la que  **$\lambda$ Page** describe y permite navegar los módulos cargados o importados, sus funciones y sus estructuras de datos para luego utilizarlas en nuestro código.

Por otra parte, hemos podido trabajar con expresiones de entrada/salida con efectos colaterales (en otras palabras, expresiones de tipo `IO a`) y hemos visto como  **$\lambda$ Page** las ejecuta sin problemas, tal como lo hace *GHCi*.

### 3.4. Otras Características de $\lambda$ **P**age

What is best in life?

---

To crush your enemies, to see them driven before you, and to hear the lamentations of their women

---

Conan, the Barbarian  
Hot water, good dentishtry and shoft lavatory paper

---

Cohen, the Barbarian

Los dos tutoriales que hemos presentado en este capítulo muestran muchas de las principales características de  $\lambda$ **P**age. Sin embargo, algunas otras características han quedado sin ser expuestas. Es nuestra intención utilizar esta sección para detallarlas, de modo que el usuario interesado pueda experimentar con ellas.

#### 3.4.1. Listas

En el caso general, cuando el usuario desea interpretar una expresión  $e$ , si esta es de un tipo que es instancia de la clase **Show**,  $\lambda$ **P**age presenta al usuario el resultado de evaluar `show e`. Ya hemos visto en el tutorial de la sección 3.3 que las expresiones de tipo **IO** a son tratadas de un modo especial (son ejecutadas y se presenta al usuario el resultado de su ejecución en caso de que sea de un tipo instancia de la clase **Show**).

Otras expresiones que son tratadas de un modo particular son las listas. Cuando el usuario decide interpretar una expresión  $e :: [a]$  tal que  $a$  es instancia de **Show**,  $\lambda$ **P**age no evalúa `show e`, sino que intenta evaluar cada elemento de la lista en forma independiente y así presentarlo al usuario. De ese modo, a pesar de que uno o más de los elementos de la lista genere una excepción al intentar ser evaluado, los demás elementos pueden ser mostrados al usuario y la interpretación no se aborta. Para entender mejor este concepto podemos observar el ejemplo de la Figura 25.

Cabe destacar que este comportamiento sólo contempla los casos en los que los elementos de la lista generen excepciones, pero no aquellos en los que su cálculo implique un cómputo infinito. Esto se debe principalmente a que no queda claro cuál debería ser el comportamiento esperado en estos casos: primeramente no hay modo de detectar cuándo un cálculo es decididamente infinito, aunque uno podría utilizar un mecanismo de timeout (de hecho,  $\lambda$ **P**age utiliza uno para las listas de caracteres, como veremos en la Sección 4.2.2); pero, más allá de ésto, en el caso de las expresiones, no siempre un cálculo infinito indica la ausencia de resultados. Por ejemplo, si la lista a mostrar es `[[1,2,3], [4..], [5,6,7]]` donde uno de los elementos es una lista que requiere un cálculo infinito pero el usuario puede observar resultados a medida que este se realiza, no es evidente que  $\lambda$ **P**age deba interrumpir su presentación para continuar con el siguiente elemento.

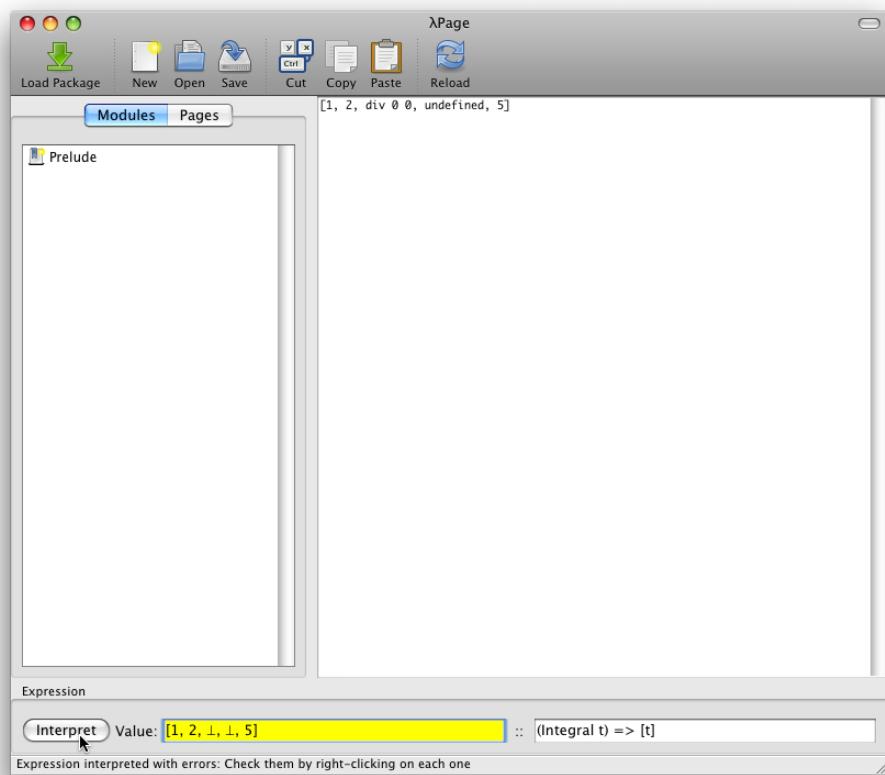


Figura 25: Interpretación de listas

En la Figura 25 podemos ver que, al evaluar una lista en la que hay dos elementos que generan excepciones,  **$\lambda$ Page** nos presenta como su interpretación el valor `[1, 2, ⊥, ⊥, 5]` y en la barra de estado nos informa que *La expresión fue interpretada con errores* y podemos chequearlos *cliqueando con el botón derecho del mouse en cada uno de ellos*. Al seleccionar el carácter  $\perp$  y hacer click con el botón derecho del mouse, se despliega el menú contextual que podemos apreciar en la Figura 26. Al presionar la opción *Explain*,  **$\lambda$ Page** nos presenta un cuadro de diálogo explicando el error, como vemos en la Figura 27.

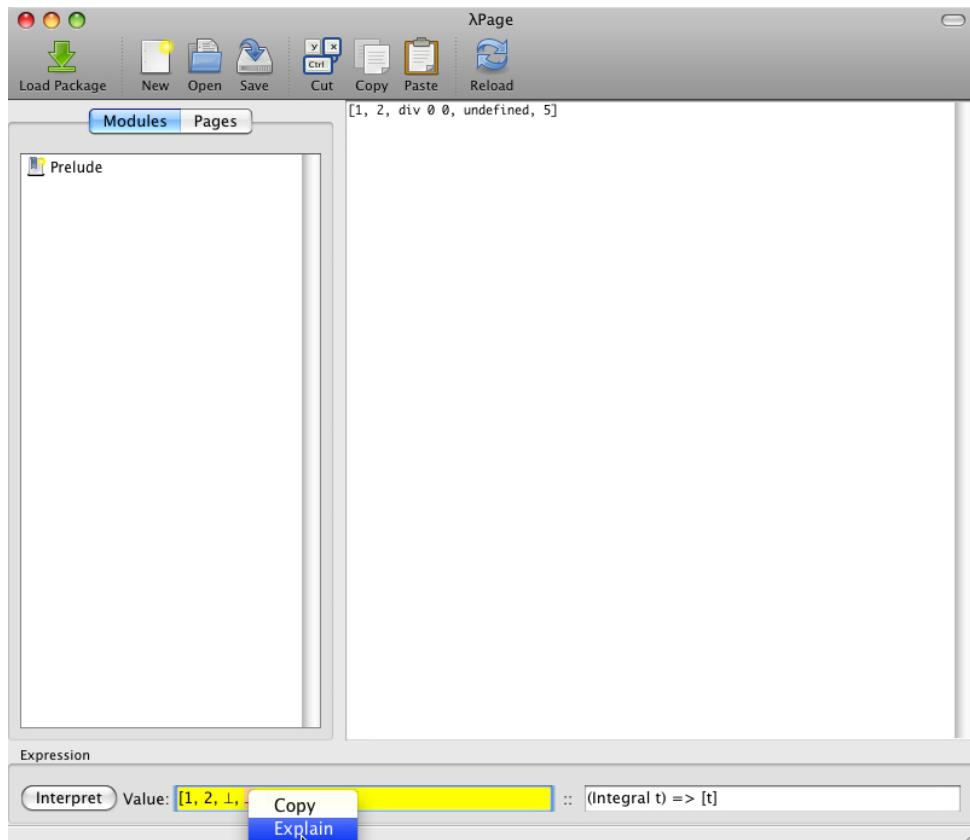


Figura 26: Interpretación de listas - Explicar

### 3.4.2. Género

Otro caso de expresiones que son tratadas de un modo particular es el de las expresiones de tipo (como pueden ser `IO ()`, `Int` o `(Num n, Monad m) => Float -> m [Float -> n]`). En estos casos,  $\lambda\text{Page}$  entiende que el usuario al presionar el botón *Interpret* no quiere conocer el valor de la expresión sino más bien el género (o, en inglés, *kind*) del tipo. En Haskell cada tipo tiene un género asociado que puede ser utilizado para asegurar que el tipo es usado de modo correcto. Las expresiones de tipos son clasificadas según sus géneros, los cuales pueden tomar una de las siguientes formas:

- El símbolo `*` representa el género de tipos asociados con datos concretos.
- Si  $k_1$  y  $k_2$  son géneros, entonces  $k_1 \rightarrow k_2$  es el género de los tipos que toman un tipo con género  $k_1$  y devuelven un tipo con género  $k_2$

Veamos como ejemplos las expresiones de las Figuras 28, 29 y 30.

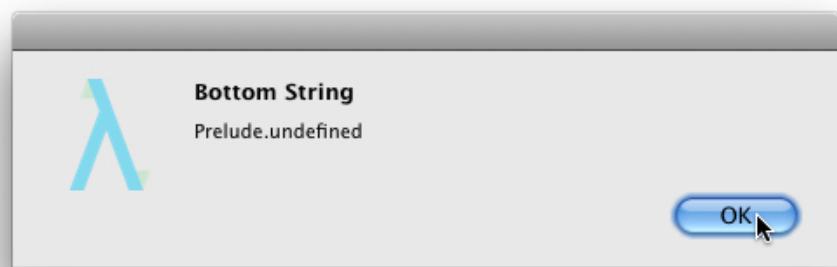


Figura 27: Interpretación de listas - Explicación

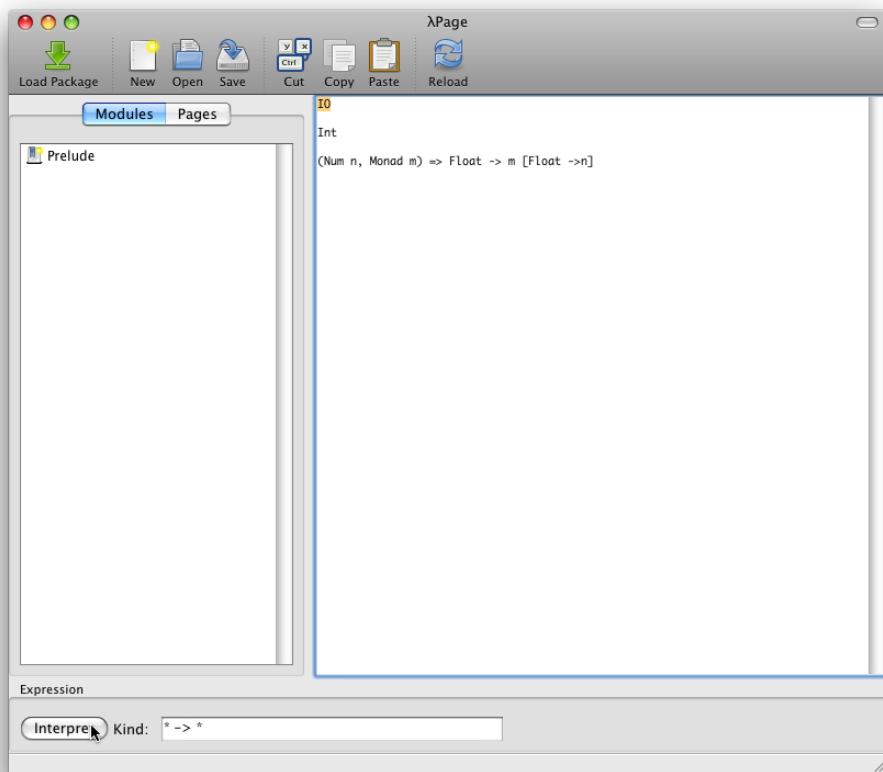


Figura 28: Géneros - IO ()

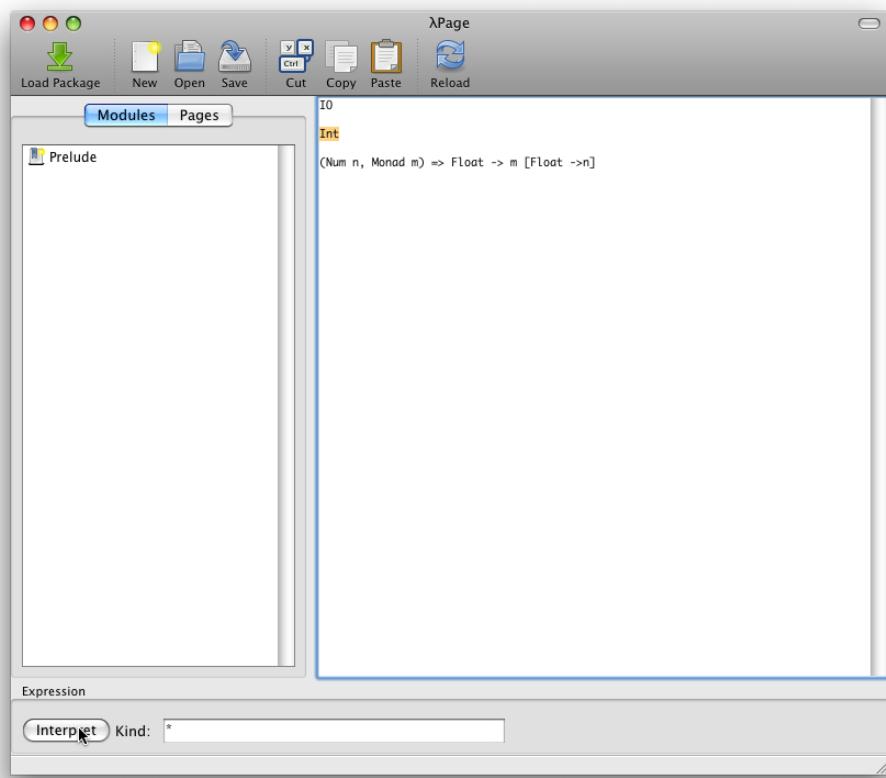


Figura 29: Géneros - Int

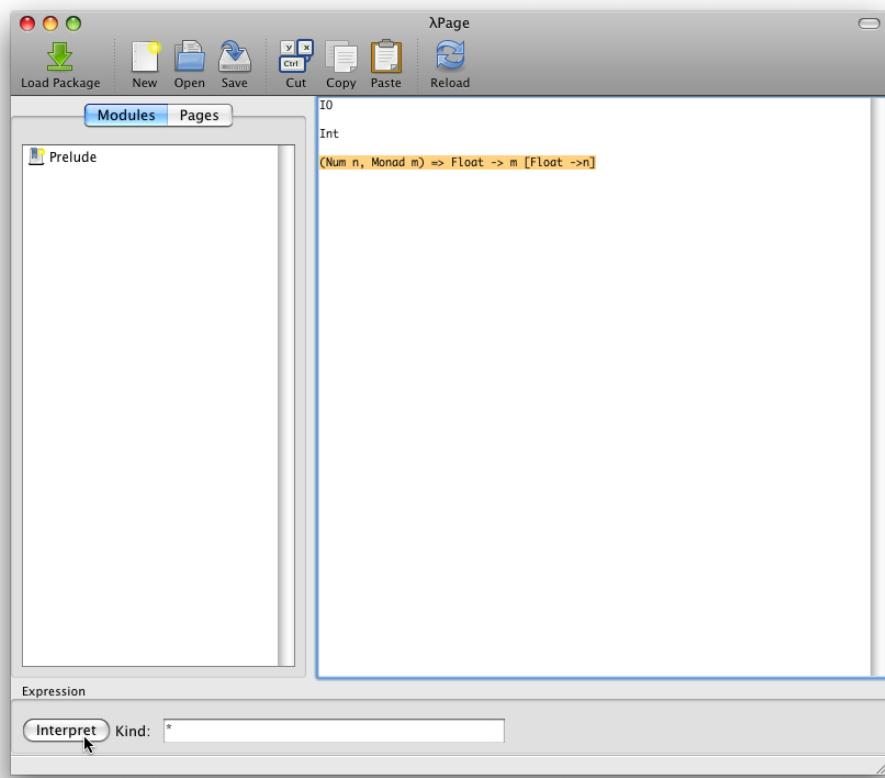


Figura 30: Géneros -  $(\text{Num } n, \text{ Monad } m) \Rightarrow \text{Float} \rightarrow m [\text{Float} \rightarrow n]$

### 3.4.3. Carga de Módulos por Nombre

A lo largo de los tutoriales de este capítulo varias veces el usuario ha debido cargar módulos y siempre lo ha hecho seleccionando el archivo con el código fuente del módulo que deseaba cargar.  **$\lambda$ Page** brinda, además de esa forma, otra alternativa para la carga de módulos: cargarlos por su nombre. Para ello, el usuario debe seleccionar la opción *Haskell → Load modules by name*. . . . El programa entonces le presentará un cuadro de texto donde ingresar los nombres de los módulos que desea cargar separados por espacios.  **$\lambda$ Page**, entonces, buscará en primera instancia el código precompilado de cada módulo e intentará cargarlo junto con los módulos de los que depende. De no ser posible, buscará el código fuente del mismo e intentará compilarlo y cargarlo. Para encontrar el código fuente,  **$\lambda$ Page** intentará ubicar el archivo correspondiente al módulo en primer lugar dentro de los directorios de archivos fuentes del paquete *Cabal* (si es que hay uno cargado) y luego en el directorio en el que se esté ejecutando la aplicación. Por ejemplo, en el caso de que no haya ningún paquete cargado y se intente cargar el módulo **Test.Server**,  **$\lambda$ Page** intentará cargar el archivo `./Test/Server.hs`.

### 3.4.4. Configuración del Compilador

Hemos mostrado en el tutorial de la Sección 3.3, cómo  **$\lambda$ Page** puede configurar su entorno y, sobre todo, las opciones necesarias para el compilador a partir de un paquete *Cabal*.  **$\lambda$ Page** permite además al usuario modificar estas opciones a través del menú *Preferences*. . . . Al seleccionar esta opción,  **$\lambda$ Page** presenta al usuario una ventana similar a la de la figura 31. En ella el usuario puede elegir las extensiones del lenguaje que desea habilitar, determinar las carpetas en las que el compilador puede ubicar los archivos con código fuente y aplicar opciones al compilador.

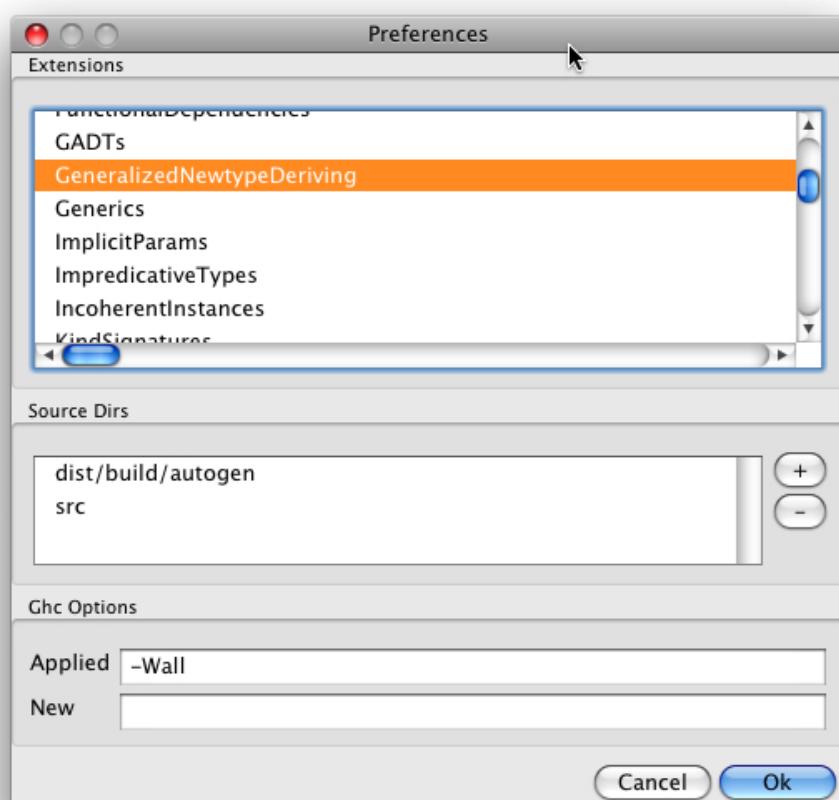


Figura 31: Opciones del Compilador

## 4. Desarrollo - ¿Cómo se hizo $\lambda$ Page?

### 4.1. Arquitectura General

If you think good architecture is expensive, try bad architecture

---

Brian Foote and Joseph Yoder

Las principales decisiones de arquitectura que se tomaron durante el desarrollo de  $\lambda$ Page tuvieron como motivaciones los siguientes requerimientos:

**Conexión con GHC**  $\lambda$ Page debía conectarse con el motor de GHC a través de su API de modo de poder detectar e interpretar expresiones. Para ello se utilizó `hint` [13]. `hint` es una librería hecha en Haskell que provee una abstracción de alto nivel sobre la API de GHC. Esta librería nos permite acceder de modo sencillo a las funciones provistas por la API y así poder evaluar expresiones, determinar su tipo y, en caso de expresiones de tipo, su género, cargar e importar módulos, utilizar extensiones del lenguaje y opciones del compilador. `hint` provee funciones que permiten este manejo y pueden ser ejecutadas dentro de la mónica `Interpreter`.

**Paralelismo**  $\lambda$ Page debía permitir al usuario editar sus páginas mientras esperaba el resultado de la evaluación de una expresión, detectar evaluaciones que podrían ser infinitas y presentar la interpretación de una expresión de manera incremental. Todas estas tareas requieren la ejecución de acciones en paralelo. Para simplificar estas tareas, se creó `eprocess` y se implementó un modelo de procesos utilizándolo. En la Sección 4.3 podremos observar en detalle cómo ha sido desarrollada esta librería.

**Errores Controlados**  $\lambda$ Page no debía fallar si la evaluación de una expresión fallaba. Más aún, también debía detectar posibles evaluaciones infinitas e informar estas situaciones al usuario. Aquí nuevamente entran en juego tanto `hint` como `eprocess`.

**Compatibilidad con GHCi**  $\lambda$ Page debía ser capaz de reemplazar a `GHCi` y, por lo tanto, brindar toda las funcionalidades que esta aplicación brinda. En particular, debía:

- ser multiplataforma
- detectar expresiones sintácticamente inválidas
- identificar el tipo de cualquier expresión sintácticamente válida
- identificar la clase de cualquier tipo sintácticamente válido
- interpretar expresiones de cualquier tipo que sea instancia de la clase `Show`
- ejecutar e imprimir el resultado de expresiones de tipo `Show a ⇒ IO a`

Teniendo en cuenta estos requerimientos, la arquitectura resultante puede ser descripta con el diagrama de la Figura 32. Esta figura presenta el estado del sistema en un instante dado. Cada bloque representan un proceso o “thread” en ejecución. Es importante no confundir estos *threads* con los threads del sistema operativo. Los “threads” utilizados por  $\lambda$ Page son administrados por la máquina virtual de GHC. Por este motivo, en un thread de sistema operativo pueden coexistir varios threads Haskell.

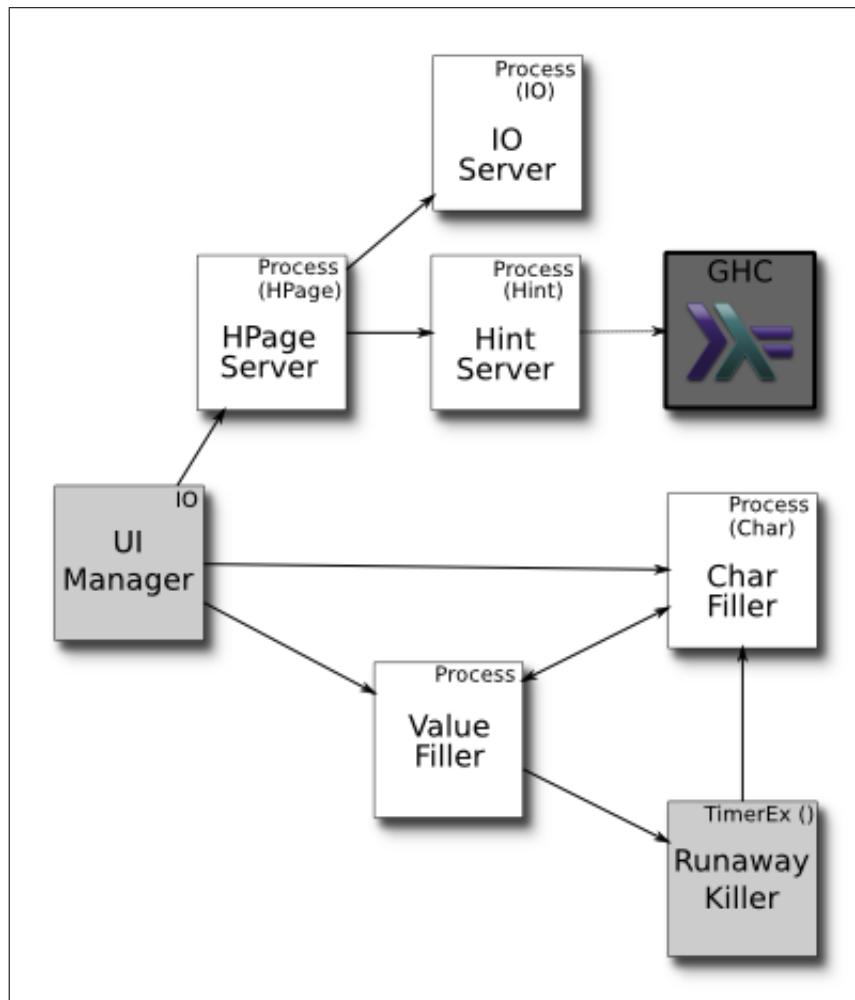


Figura 32: Arquitectura de  $\lambda$ Page

Cada uno de estos procesos se ejecuta dentro del entorno de una mónada, la cual se encuentra identificada en la esquina superior derecha del bloque. En el diagrama podemos identificar los siguientes componentes:

**UI Manager** Este es el thread que inicia el programa, genera y administra la interfaz del usuario utilizando las herramientas provistas por *wxHaskell*. En este thread se mantiene el estado visual de la aplicación: el estado de los controles, la última búsqueda realizada, etc.

**HPage Server** Este proceso, iniciado por el **UI Manager**, es el que comunica a la interfaz del usuario con la máquina virtual de *GHC*, a través del **Hint Server**. En el caso de expresiones de tipo **IO a**, este mismo proceso se comunica con el **IO Server** para ejecutarlas y obtener su resultado o capturar sus errores. En este proceso se mantiene el estado general de la aplicación: sus páginas, expresiones, paquetes y módulos cargados, etc. Este proceso permite la ejecución de acciones definidas en la mónada **HPage** de manera sincrónica o asincrónica. En el caso de acciones ejecutadas asincrónicamente, permite que las mismas sean canceladas. En tales casos, dado que la API de *GHC* no provee un mecanismo para cancelar la acción en curso, **HPage Server** reinicia el **Hint Server** y se encarga de “ponerlo al día” aplicando las acciones necesarias para que se encuentre en el estado anterior a la ejecución de la última acción.

**IO Server** Este proceso, iniciado por el **HPage Server**, es el encargado de ejecutar acciones de tipo **IO a** en un entorno controlado y obtener su resultado.

**Hint Server** Este proceso, iniciado por el **HPage Server**, mantiene una conexión con la máquina virtual de *GHC* a través de su API

**Value Filler** Este proceso, iniciado por el **UI Manager** es el encargado de procesar el resultado obtenido del **HPage Server**. Cabe recordar aquí que *Haskell* trabaja con evaluación “lazy”, por lo cual el resultado obtenido no ha sido aún completamente procesado. El **Value Filler** espera recibir un resultado y, al recibirla, se encarga de evaluarlo y mostrarlo por pantalla, para ello envía y recibe mensajes del **Char Filler** a fin de procesar cada carácter a mostrar.

**Char Filler** Este proceso, iniciado por el **UI Manager** cumple una muy sencilla función: utilizando los procedimientos de envío y recepción de mensajes provistos por *eprocess*, espera recibir un carácter (o sea, una expresión de tipo Char), para luego evaluarlo y enviar como respuesta su valor en forma normal.

**Runaway Killer** Este thread, instancia del tipo *TimerEx* provisto por *wxHaskell*, es iniciado por cada **Value Filler** al momento de enviar un nuevo carácter al **Char Filler**. El objetivo del **Runaway Killer** es el de detectar procesamiento “posiblemente” infinito. Básicamente, pasado un segundo de procesamiento, reinicia el **Char Filler** e informa al **Value Filler** que lo inició que el carácter que se esperaba procesar ha demorado demasiado y podría desencadenar una evaluación infinita.

Para poder entender el funcionamiento de los distintos procesos con un mayor detalle, presentaremos un ejemplo en el cual el usuario define la expresión `e = [1, undefined, 3, div 0 0] ++ [5..]` e intenta interpretarla, pero, mientras `λPage` está presentándole los resultados, decide cancelar la interpretación. Elegimos la expresión `e` por ser una lista infinita y además presentar algunos elementos que generan distintos tipos de excepciones. La Figura 33 nos muestra el diagrama de secuencia correspondiente al ejemplo.

Podemos ver en el diagrama que el **HPage Server**, al detectar que la expresión a evaluar es una lista, en lugar de solicitar al **Hint Server** que evalúe la expresión original, le solicita la interpretación de la expresión `map show e`, de modo de obtener una lista de `Strings` para retornar al **UI Manager**. Luego, el **Value Filler**, al recibir una lista de valores a mostrar, intenta presentar cada uno de ellos por separado, concatenándolos del

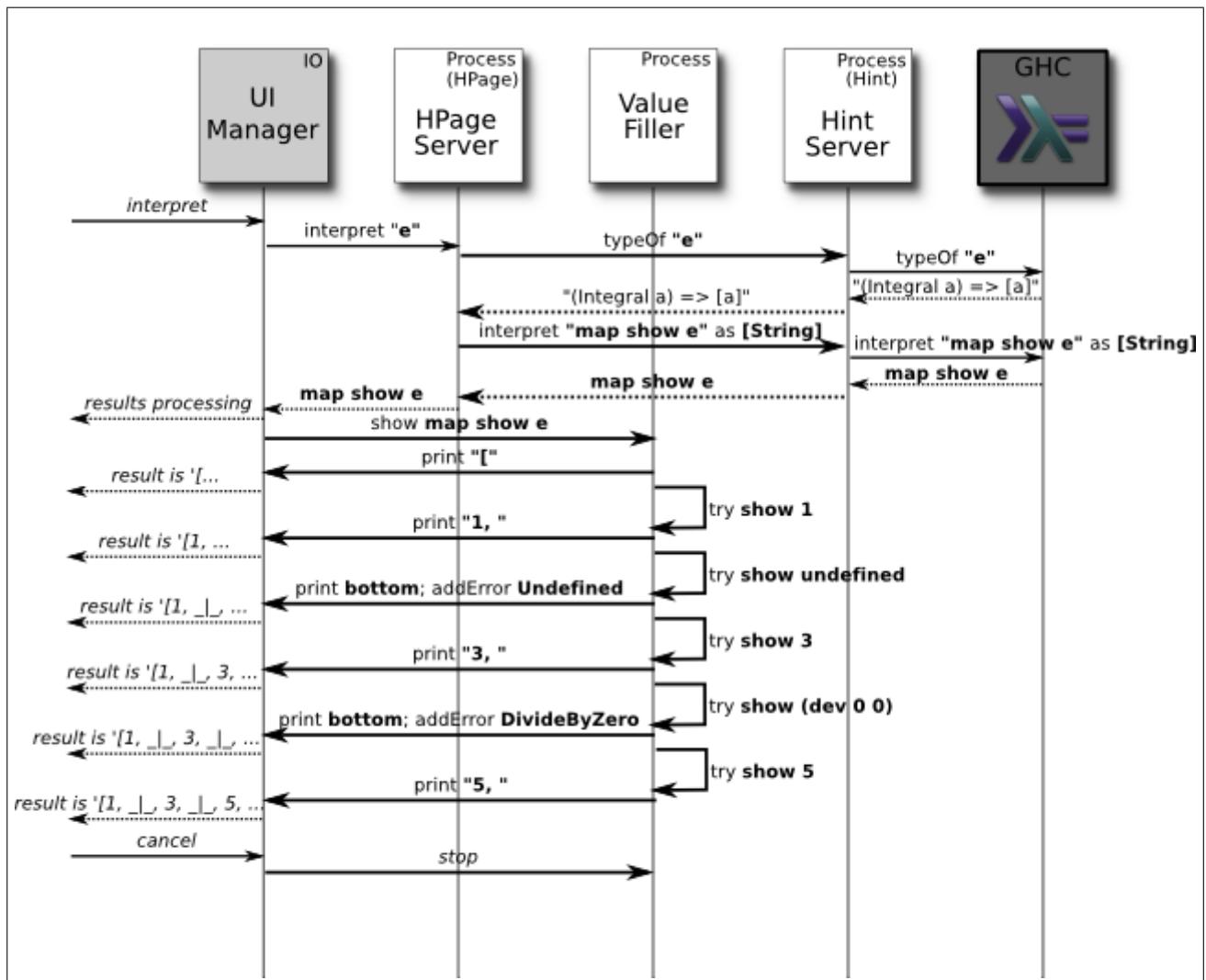


Figura 33: Secuencia de Evaluación de `e = [1, undefined, 3, div 0 0] ++ [5, ...]`

modo en que generalmente se presentan las listas en *Haskell*. Cuando el usuario decide cancelar, **HPage Server** detiene el **Value Filler** y, por lo tanto, este proceso se interrumpe.

Con el objetivo de destacar otras interacciones más significativas, en el diagrama de la Figura 33 hemos omitido la parte del proceso que involucra al **Char Filler** y al **Runaway Killer**. Para visualizar mejor su utilidad, crearemos un tipo de datos un tanto “artificial” que llamaremos **WithInfiniteChar** y definiremos a continuación. Veremos en la Figura 34 un diagrama de secuencia correspondiente a un proceso de evaluación de la expresión **WIC**.

```
data WithInfiniteChar = WIC

instance Show WithInfiniteChar where
    show WIC = [ 'c' , head . show $ length [ 1 .. ] ]
```

Como puede observarse, al intentar mostrar la expresión **WIC**,  **$\lambda$ Page** se encontrará con una cadena cuyo segundo carácter no puede computar pues requiere un cálculo infinito, en la Figura representamos este carácter con la letra  $\Omega$ . Allí es donde entra en acción el **Runaway Killer** para informar esta situación al usuario.

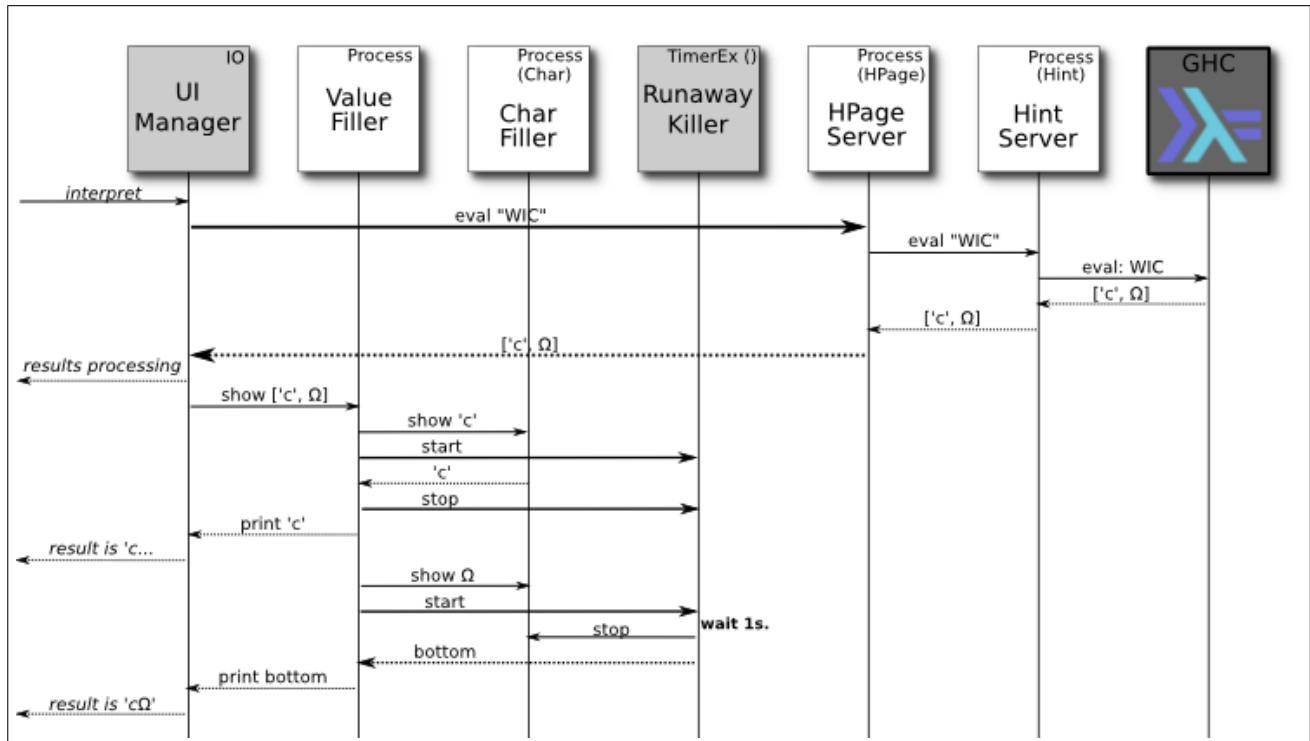


Figura 34: Secuencia de Evaluación de WIC

## 4.2. Diseño

Design and programming are human activities;  
forget that and all is lost

---

Bjarne Stroustrup

Presentaremos a continuación las principales decisiones de diseño que se han tomado durante la creación de  $\lambda\text{Page}$ . Todas ellas tienen como fundamento los requerimientos principales exhibidos en la sección anterior y también algunos requerimientos adicionales, como la integración con *Cabal* y *Hayoo!*.

### 4.2.1. Concurrencia

Como hemos visto en la sección anterior, al momento de diseñar  $\lambda\text{Page}$  tuvimos que considerar la necesidad de paralelizar tareas, para permitir al usuario, por ejemplo, trabajar en un documento mientras el motor de  $\lambda\text{Page}$  evalúa una expresión. También debemos considerar que estas tareas a realizar en paralelo no son totalmente independientes sino que requieren una sincronización. Tomando la idea del modo en que está diseñado el lenguaje de programación *Erlang*, decidimos implementar el paralelismo utilizando lo que denominamos procesos. Conceptualmente, los procesos son hilos de ejecución que se realizan en paralelo y pueden recibir o enviar mensajes. Esta característica de mensajería entre procesos es la que permite la sincronía cuando es necesaria. Por otra parte, a diferencia de *Erlang*, al utilizar *Haskell*, los mensajes enviados de un proceso a otro pueden ser mucho más complejos. Gracias al hecho de que en *Haskell* las acciones son elementos de primer orden, un proceso puede enviar a otro directamente las acciones que desea que éste ejecute, tal como deben ser ejecutadas. Esta es una característica esencial para reducir la complejidad de la implementación de todos nuestros procesos, en particular de aquellos que actúan como servidores (**IO Server**, **HPage Server** y **Hint Server**), como veremos luego en la Sección 4.3.

### 4.2.2. Bottoms

El lenguaje *Haskell* tiene una característica única: *la evaluación perezosa* o *lazy evaluation*. Gracias a esta característica, las expresiones *Haskell* no son completamente evaluadas (reducidas a forma normal) hasta el momento en que realmente se necesita conocer su valor. Dado que  $\lambda\text{Page}$  presenta al usuario un interprete de expresiones, es necesario que esté preparado para no sólo soportar sino también aprovechar esta característica. En particular, dentro de  $\lambda\text{Page}$  las expresiones son reducidas a forma normal al momento de intentar mostrar el resultado de su evaluación al usuario. En ese momento,  $\lambda\text{Page}$  distingue cuatro tipos de valores:

**Entrada/Salida** Expresiones cuyo valor es de tipo `IO a`, donde `a` es un tipo que es instancia de la clase `Show`. En este caso,  $\lambda\text{Page}$  intentará ejecutar la acción, obtener su resultado y presentarlo al usuario como se verá a continuación.

**Listas** Expresiones cuyo valor es de tipo `[a]`, donde `a` es un tipo que es instancia de la clase `Show`. En este caso,  $\lambda\text{Page}$  intenta reducir cada elemento e ir presentándolo al usuario dentro de una lista. Cada elemento entonces, es tratado como se verá a continuación.

**Expresiones “visibles”** Expresiones cuyo tipo es instancia de la clase `Show`. `λPage` intenta evaluar la expresión a mostrar y pueden suceder varias cosas:

- Por supuesto, puede suceder que al evaluar la expresión se obtenga una cadena de caracteres, en cuyo caso `λPage` simplemente presentará el resultado al usuario
- Puede suceder que, intentando evaluar la expresión se obtenga una cadena de caracteres de longitud infinita. La política de `λPage` en este caso es permitir al usuario decidir cuándo desea abortar la evaluación y mostrar la porción del resultado obtenida hasta ese momento
- Puede suceder también que, intentando evaluar un carácter, se genere un cálculo “infinito” o una excepción. En este caso también, `λPage` permite al usuario cancelar la evaluación cuando lo considere apropiado.
- Otra posibilidad es que, luego de presentar un carácter, al intentar obtener el resto de la cadena, `λPage` encuentre un cálculo “infinito”. En estos casos, nuevamente `λPage` permite al usuario cancelar la evaluación cuando lo considere apropiado.
- Por último, es posible que, luego de presentar un carácter, al intentar obtener el resto de la cadena, `λPage` encuentre una excepción. En estos casos, `λPage` informa la situación al usuario y aborta la evaluación de la expresión.

**Expresiones no “visibles”** Expresiones cuyo tipo no es instancia de la clase `Show`. `λPage` ante estas expresiones, se limita a mostrar su tipo.

La distinción que hace `λPage` ante los distintos tipos de expresiones a procesar y el comportamiento escogido al momento de mostrar el resultado de su evaluación tienen como objetivo brindar al usuario la mayor cantidad de información posible sobre la expresión que desea interpretar. Sin ellos, el comportamiento natural de un programa como `λPage` ante una expresión `e`, sería el de asignar el resultado de `show e` al cuadro de texto que presenta el resultado al usuario.

En el caso de las acciones de entrada/salida, `λPage` al igual que `GHCi`, considera que el usuario no está simplemente interesado en conocer el tipo de las acciones, sino en ejecutarlas para producir (de ser preciso) ciertos efectos secundarios y obtener un resultado. Es razonable pensar que quien construye una acción de este tipo (que no es instancia de la clase `Show`) pero que produce (al ser evaluada) un resultado de un tipo que sí lo es, está interesado en ejecutar la acción y observar el resultado.

En otro caso, observemos con un ejemplo, qué sucedería si `λPage` asignase directamente el resultado de `show e` al cuadro de texto. Para nuestro ejemplo, vamos a crear un tipo de datos similar al utilizado para el diagrama de secuencia de la Figura 34 que nos va a permitir mostrar todos los casos posibles:

```
data WithInfiniteChar = WIC Int

instance Show WithInfiniteChar where
    show (WIC x) | x < 0 = [ 'c' , head $ show x]
                  | x == 0 = undefined
                  | otherwise = [ 'c' , head . show $ length [1..] ] 

test = map WIC [-5,-4..]
```

Si el usuario quisiese interpretar la expresión `[-5, -4..]` y  $\lambda\text{Page}$  intentase asignar `show` `[-5, -4..]` al cuadro de texto, haría falta reducir a forma normal una lista infinita, cálculo que nunca terminaría y por tanto, el usuario no vería ningún resultado en la pantalla.

Si, en su lugar, el usuario quisiese interpretar la expresión `test`, el intérprete generaría una excepción `Prelude.undefined` al intentar mostrar el sexto elemento de la lista (el correspondiente a 0 en la lista original) y esa excepción sería todo lo que el usuario podría saber sobre la evaluación de su expresión.

Entonces, en vez de asignarlo directamente,  $\lambda\text{Page}$  podría evaluar la expresión carácter a carácter e ir presentando cada uno de ellos al usuario a medida que los va obteniendo. En ese caso, los caracteres correspondientes a los primeros 5 elementos de la lista (`[c-, c-, c-, c-, c-, ]`) podrían ser mostrados al usuario, pero al llegar al sexto, nuevamente el intérprete generaría la excepción y ya ningún otro elemento de la lista podría ser mostrado. Aproximadamente de este modo es como se comportan tanto *GHCi* como *Hugs98*.

Al diseñar  $\lambda\text{Page}$  decidimos intentar dar un resultado aún más completo. Para ello,  $\lambda\text{Page}$  puede observar que la expresión `test` tiene tipo `[WithInfiniteChar]` y ver que se trata de una lista cuyos elementos son de un tipo que es instancia de la clase `Show`.  $\lambda\text{Page}$  “sabe” cómo se representan las listas y, por lo tanto, podría emular el comportamiento de `show` en esos casos. Lo que  $\lambda\text{Page}$  podría hacer pues es intentar aplicar `show` a cada elemento por separado, mostrar (de modo análogo a como se presentan las listas) aquellos que efectivamente puedan ser evaluados y exhibir de otro modo aquellos que no. Para estos casos, en los que por algún motivo, un elemento no puede ser mostrado, hemos elegido presentar el carácter  $\perp$  y permitir al usuario, a través de un menú contextual saber por qué no se ha podido mostrar el elemento. En nuestro ejemplo, con esta modalidad,  $\lambda\text{Page}$  presentaría al usuario una lista que comenzaría con `[c-, c-, c-, c-, c-,  $\perp$ ,  $c\perp$ , c]` y, al llegar al séptimo elemento, dado que el intérprete debería resolver un cálculo infinito para presentar el siguiente carácter, dejaría de presentar resultados y el usuario no tendría otra alternativa que presionar el botón *Cancel* para detener la ejecución.

La versión actual de  $\lambda\text{Page}$  tiene en cuenta estos casos de cálculos infinitos y, por ello, al ir evaluando carácter por carácter, verifica que ninguno de ellos requiera más de un segundo en ser calculado. En caso de demorar más que ello,  $\lambda\text{Page}$  introducirá el carácter  $\perp$  y, por lo tanto, presentará al usuario la lista, en principio, infinita que comienza con `[c-, c-, c-, c-, c-,  $\perp$ ,  $c\perp$ ,  $c\perp c$ ,  $c\perp c$  ...]`

De este modo,  $\lambda\text{Page}$  intenta mostrar al usuario tanta información como sea posible sobre la expresión que se quiere interpretar. Está claro que, si bien son probablemente los más habituales, las acciones de entrada/salida y las listas son sólo dos casos especiales de un gran conjunto de tipos de expresiones cuya interpretación puede ser presentada de modo de brindar mayor información que la que brinda la evaluación directa de `show`. Nos explayaremos más sobre este tema y sobre cómo  $\lambda\text{Page}$  podría adaptarse para contemplar otros casos en la Sección 5.2.

#### 4.2.3. Integración

Una de las herramientas más comúnmente usadas por los desarrolladores haskell es *Cabal*. En un paquete *Cabal*, el desarrollador define los módulos que componen su aplicación o librería, los lugares (carpetas) dónde encontrar el código fuente y los recursos que éstos necesitan para funcionar, junto con las extensiones que se requieren para poder compilarlos.

*λPage* por su parte, permite al desarrollador cargar o importar módulos para poder utilizarlos al momento de evaluar expresiones. También permite definir los lugares donde el compilador puede encontrar archivos fuentes y las extensiones que éste debe utilizar al momento de compilar los archivos encontrados.

Observando estas similitudes, una integración con *Cabal* es algo que surge de manera natural y *λPage* lo provee. *λPage* permite al desarrollador cargar un paquete *Cabal* previamente configurado y de ese modo utilizar los módulos, extensiones y ubicaciones en él definidos.

Otra herramienta, quizá no tan popular como *Cabal*, pero también muy útil es *Hayoo!*. Conociendo esta herramienta, decidimos integrarla con *λPage* de modo que el desarrollador pueda realizar consultas en su base de datos para obtener información sobre alguna función, tipo, módulo, clase o expresión que desee analizar.

## 4.3. Implementación

Nothing resolves design issues like an implementation

---

J. D. Horton

A child of five would understand this. Send someone to fetch a child of five

---

Groucho Marx

### 4.3.1. eprocess

Para la implementación de `eprocess`, nuestra librería de procesos, utilizamos varias herramientas de paralelismo y concurrencia que se encuentran muy bien descriptas en el libro [Real World Haskell](#) [34]. Utilizamos *Threads* para paralelizar procesos y *Channels* y *MVars* para permitirles comunicarse.

Un *Thread* es una acción de entrada/salida que se ejecuta de manera independiente. Para crear un *Thread*, se debe importar el módulo `Control.Concurrent` y utilizar la función `forkIO`.

Una *MVar* representa una *caja para un único elemento*: puede estar llena o vacía. Podemos poner algo en la caja, llenándola, o sacarlo, vaciándola. Si intentamos poner algo en una *MVar* que ya está llena, nuestro *Thread* quedará bloqueado hasta que otro tome su contenido y, por ende, la vacíe. Del mismo modo, si intentamos tomar el valor de una *MVar* que está vacía, nuestro *Thread* esperará hasta que alguien ponga un valor en ella.

Finalmente, un *Channel* es una abstracción de una vía de comunicación unidireccional o, visto de otro modo, una cola de mensajes. Siempre se puede agregar un nuevo mensaje a un canal sin bloquear el *Thread* que escribe. Sin embargo, si el canal está vacío, el *Thread* que intente leer se bloqueará hasta que llegue el primer valor.

Con estos elementos, definimos entonces un tipo monádico para representar a las acciones a realizarse en procesos paralelos de tipo `m`, tales que retornan expresiones de tipo `a` y pueden recibir elementos de tipo `r`:

```
newtype ReceiverT r m a = RT { internalReader :: ReaderT (Handle r) m a }
    deriving (Monad, MonadIO, MonadTrans, MonadCatchIO)
```

Individualizamos luego este tipo genérico, definiendo el tipo `Process`:

```
type Process r = ReceiverT r IO
```

Finalmente definimos las funciones que permiten la ejecución y mensajería entre procesos:

```
spawn :: MonadIO m => Process r k -> m (Handle r)
kill :: MonadIO m => Handle a -> m ()
self :: Monad m => ReceiverT r m (Handle r)
sendTo :: MonadIO m => Handle a -> a -> m ()
recv :: MonadIO m => ReceiverT r m r
```

Las funciones `spawn` y `kill` inician y detienen un proceso respectivamente. `spawn` retorna como resultado un `Handle`, que identifica únicamente al proceso y permite enviarle mensajes utilizando la función `sendTo`. Cabe notar que esta función no necesita ser utilizada dentro de un `process` sino simplemente en cualquier instancia de la clase `MonadIO`. De este modo, el `thread` que haya ejecutado `spawn` puede ejecutar `sendTo` para enviar mensajes al proceso que ha iniciado.

Luego, dentro de una instancia de `ReceiverT` se puede utilizar la función `self` para conocer el propio `Handle` y `recv` para recibir mensajes. Cabe notar que `recv` se ejecuta de manera bloqueante, emulando el comportamiento de la función `receive` de *Erlang*. Sin embargo, a diferencia de *Erlang*, el tipado estático de *Haskell* permite garantizar en tiempo de compilación que ningún proceso reciba un mensaje que no está esperando. Para notar esto basta observar que en el tipo de la función `sendTo` el parámetro del tipo `Handle` que indica el tipo de mensajes que espera recibir el proceso y el tipo del mensaje a enviar deben coincidir.

#### 4.3.2. Servers

Con el objetivo de separar la interacción con GHC del resto de la ejecución del sistema y de esta manera, poder capturar sus excepciones y aislarlas, hemos encapsulado la ejecución de estas acciones (correspondientes a la mónada **Interpreter**) en un proceso particular, al que denominamos **Hint Server**. Por otra parte, con objetivos similares, decidimos aislar la ejecución de las acciones propias de  $\lambda$ **Page** (correspondientes a la mónada **HPage**) de las relativas a la interfaz de usuario, para lo cual creamos el proceso **HPage Server**. A su vez, también fue necesario ejecutar acciones de la mónada **IO** en un contexto controlado, por lo que desarrollamos el proceso **IO Server**. Gracias al uso de mónadas, al hecho de que las acciones sean objetos de primer tipo y a los mecanismos provistos por **eprocess**, pudimos construir estos servers de una manera sencilla. Mostramos, a modo de ejemplo, el código del **IO Server**:

```
module HPage.IOServer (start, stop, runIn, ServerHandle) where

import Control.Exception (try, SomeException)
import Control.Monad
import Control.Monad.Trans
import Control.Concurrent.Process

newtype ServerHandle = SH {handle :: Handle (IO ())}

start :: IO ServerHandle
start = spawn ioRunner >>= return . SH
    where ioRunner = forever $ recv >>= liftIO

runIn :: ServerHandle -> IO a -> IO (Either SomeException a)
runIn server action = runHere $ do
    me <- self
    sendTo (handle server) $ try action >>=
        sendTo me
    recv

stop :: ServerHandle -> IO ()
stop = kill . handle
```

Cabe destacar la simpleza de este módulo que, con no más de 20 líneas de código es capaz de ejecutar cualquier acción dentro de la mónada **IO** en un proceso aislado y controlado.

Para entender el comportamiento de este módulo, iremos paso a paso:

**Imports** Al inicio del módulo se realiza la importación de los siguientes módulos auxiliares:

**Control.Exception** Se importan la función **try** para poder capturar las excepciones que se generen durante la ejecución de las acciones y el constructor **SomeException** pues es el constructor del tipo de excepción más genérico, el mismo engloba a todos los demás. Nuestro objetivo es poder capturar cualquier excepción que se genere durante la ejecución de una acción.

**Control.Monad** Se importa el módulo para poder utilizar las funciones `>>` y `>>=` que permitirán concatenar acciones, la función `return` para determinar explícitamente el resultado de una acción y la función `forever` que explicaremos más adelante.

**Control.Monad.Trans** Se importa el módulo para poder combinar acciones de tipo `Process` con acciones de tipo `IO` a utilizando la función `liftIO`

**Control.Concurrent.Process** Este es el módulo en el que se encuentran definidas las funciones de la librería `eprocess`

**ServerHandle** Luego se define el tipo `ServerHandle`, cuyas instancias representan identificadores únicos de `IO` Servers. El proceso que inicie un servidor obtendrá un `ServerHandle` que le permitirá comunicarse con él.

**start** A continuación se define la función que inicia el servidor y retorna su `ServerHandle`. El servidor es iniciado utilizando la función `spawn` provista por `eprocess`, la cual retorna un `Handle` que es encapsulado en un `ServerHandle` utilizando su constructor `SH`. La función `spawn` toma como primer parámetro la acción (de tipo `Process`) que se ejecutará en el nuevo proceso. En este caso, esa acción recibe el nombre de `ioRunner` y se define utilizando la función `forever` que ejecuta, en principio, indefinidamente la acción que recibe por parámetro. En nuestro caso, la acción a ejecutar indefinidamente es `recv >>= liftIO`, la cual puede leerse como “esperar a recibir un valor y, una vez recibido, aplicarle la función `liftIO`”. Esta función permite ejecutar acciones de tipo `IO` a dentro del contexto de otra mónada, en este caso `Process` a.

**runIn** La siguiente función que aparece en el módulo es `runIn`, una función pensada para ejecutar acciones. Esta función toma como parámetro un `ServerHandle` y una acción (de tipo `IO a`), compone una nueva acción (de tipo `IO ()` que consiste en intentar evaluar la acción recibida y luego enviar el resultado de la misma nuevamente al proceso en el que se está evaluando `runIn`), envía esta nueva acción al servidor correspondiente al `ServerHandle` recibido y se bloquea esperando el resultado. Observemos en detalle las funciones involucradas en este proceso:

**runHere** Esta función permite ejecutar acciones de tipo `Process a` en el contexto de la mónada `IO`.

**self** Como hemos visto en la sección 4.3.1, esta función nos permite conocer el `Handle` del propio proceso.

**sendTo** Utilizando esta función se envía a un proceso un mensaje, en este caso la acción modificada que se espera ejecute.

**handle** Esta función, definida de forma implícita en el constructor del tipo `ServerHandle` nos devuelve el `Handle` del mismo, que es lo que `sendTo` necesita para “ubicar” al proceso al que debe enviar el mensaje.

**try** Esta función intenta ejecutar una acción y devuelve, en caso de detectar una excepción, la expresión `Left e` donde `e` es la excepción capturada y, en caso de que la acción se ejecute correctamente, la expresión `Right a` donde `a` es el resultado de la acción.

**>>=** Esta función toma una acción de algún tipo que sea instancia de la clase `Control.Monad` y una función y genera una nueva acción que consiste en evaluar la primera, luego construir un nueva acción utilizando la función con el resultado obtenido como parámetro y finalmente ejecutar esta segunda acción. En nuestro caso, la utilizamos para que el servidor (usando `sendTo me`) envíe al proceso ejecutado con `runHere` el resultado de `try action`.

**recv** Como hemos visto en la sección 4.3.1, esta función bloquea al proceso en espera de recibir algún mensaje y retorna ese mensaje una vez recibido.

**stop** La última de las funciones simplemente compone las funciones `handle` (nos devuelve el `Handle` del servidor) con `kill` que, como hemos visto en la sección 4.3.1, lo detiene.

#### 4.3.3. Módulos de $\lambda$ Page

El módulo principal de la aplicación es el denominado `HPage.Control`. Este módulo describe la monada `HPage` e incluye todas las acciones que pueden realizarse en ella. Es el encargado de mantener el estado del sistema, para lo cual hemos definido un tipo de datos llamado `Context`, que mostraremos a continuación.

```
newtype Expression = Exp {exprText :: String}
    deriving (Eq, Show)

data InFlightData = LoadModules { loadingModules :: Set String ,
                                    runningAction :: Hint.InterpreterT IO () }
                    |
                    ImportModules { importingModules :: Set String ,
                                     runningAction :: Hint.InterpreterT IO () }
                    |
                    SetSourceDirs { settingSrcDirs :: [FilePath] ,
                                     runningAction :: Hint.InterpreterT IO () }
                    |
                    SetGhcOpts { settingGhcOpts :: String ,
                                 runningAction :: Hint.InterpreterT IO () }
                    |
                    Reset

data Page = Page { — Display —
                  expressions :: [Expression] ,
                  currentExpr :: Int ,
                  undoActions :: [HPage ()] ,
                  redoActions :: [HPage ()] ,
                  — File System —
                  original :: [Expression] ,
                  filePath :: Maybe FilePath
                }

data Context = Context { — Package —
                        activePackage :: Maybe PackageIdentifier ,
                        pkgModules :: [Hint.ModuleName] ,
                        — Pages —
                        pages :: [Page] ,
                        currentPage :: Int ,
                        — GHC State —
                      }
```

```

    loadedModules :: Set String ,
    importedModules :: Set String ,
    extraSrcDirs :: [ FilePath ] ,
    ghcOptions :: String ,
    server :: HS.ServerHandle ,
    ioServer :: HPIO.ServerHandle ,
    -- Actions --
    running :: Maybe InFlightData ,
    recoveryLog :: Hint.InterpreterT IO ()
}

```

Los principales componentes del estado del sistema son:

**activePackage** El paquete *Cabal* activo, si es que se ha cargado alguno.

**pages** Las páginas que el usuario está viendo. Cabe notar que un usuario puede tener varias páginas activas a la vez, una con cada documento.

**loadedModules / importedModules** Los módulos que el usuario ha cargado / importado

**server** El handle del **Hint Server** que el mismo **HPage Server** inicia y mantiene

**ioServer** El handle del **IO Server** que el mismo **HPage Server** inicia y mantiene

**running** La acción que se encuentra ejecutándose de manera asincrónica, si es que hay alguna

**recoveryLog** El log de acciones realizadas hasta el momento, necesario al momento de detener el **Hint Server**.

Cabe destacar aquí cómo se construye este log: Se trata de una única acción de tipo `Hint.InterpreterT IO ()` que se va construyendo de manera incremental al realizar cada acción que involucra a `Hint` en la mánada `HPage`. Cuando una acción es aplicada (o sea, al momento en que ya finalizó su ejecución - recordemos que `HPage.Control` permite ejecutar acciones de manera sincrónica o asincrónica-), se ejecuta el siguiente código para agregarla al log (donde `c` es el contexto actual, `ra` es la acción ejecutada y el resultado es el nuevo contexto actual, una vez almacenada la acción):

```
c{ recoveryLog = (recoveryLog c) >> ra >> return () }
```

Nótese que el paquete *Cabal*, las extensiones y los módulos cargados o importados, etc. son independientes de las páginas con las que el usuario esté trabajando, por lo que el usuario por ejemplo no puede manejar dos paquetes *Cabal* a la vez. Ésto se debe a que la API de *GHC* no permite la utilización de *multi-threading*, por lo tanto, dentro de un programa sólo puede haber una única lista de módulos cargados/importados, una única lista de extensiones, etc.

También debe notarse que mucha de la información de estado guardada en el **Context** es “redundante” pues se configura directamente en el **Hint Server**. Eso se debe a que ante la necesidad de reiniciarlo, el **HPage Server** restaura su estado utilizando esos datos.

Finalmente, `running` y `recoveryLog` se utilizan para permitir al usuario cancelar acciones que intenta ejecutar de modo asincrónico. Cada acción que se desea realizar de forma asincrónica, devuelve una `MVar` que se llenará en caso de finalizar la acción con éxito y se acumulará en el `recoveryLog`. En caso de que el usuario decida cancelar, el **HPage Server** reiniciará el **IO Server** y el **Hint Server**, configurará a este último según los demás parámetros (ej: `ghcOptions`) y ejecutará `recoveryLog` para “ponerlo al día”.

El estado de las páginas con las que el usuario trabaja está definido como una lista de expresiones y dos listas de acciones para permitir el uso de *undo* y *redo*. Finalmente, si la página corresponde a un archivo en disco, `λPage` identifica el “path” del mismo para poder guardarlo o recargarlo de ser necesario.

Gracias a haber separado la lógica correspondiente a la interfaz de usuario y la propia de `λPage`, hemos podido desarrollar esta última utilizando la técnica de TDD (Test Driven Development) [1]. Para ello utilizamos `QuickCheck` [26], una herramienta de testeo automático para programas Haskell que nos permitió ir desarrollando y verificando tests de manera incremental hasta alcanzar la actual definición del **HPage Server**. Los tests desarrollados pueden ser ejecutados utilizando el siguiente comando:

```
$ cabal test hpage
```

#### 4.3.4. UI

La interfaz gráfica de `λPage` está desarrollada utilizando `wxHaskell`, un framework elegido por ser multiplataforma y, gracias a estar construido sobre `wxWidgets`, presentar un “look&feel” nativo en distintos entornos. `wxHaskell` es un framework sencillo para utilizar y entender y, pese a que aún se encuentra en período de evolución, es suficientemente estable. Sin embargo, tal como puede verse en `wxhNotepad` hemos tenido que superar varios escollos hasta lograr una UI estable e intuitiva. A los lectores interesados en estos detalles técnicos les recomendamos los artículos escritos por Jeremy O'Donoghue en su tutorial `Building a text editor` [32].

## 5. Resultados

### 5.1. Objetivos Alcanzados

Results! Why, man? I have gotten a lot of results. I know several thousand things that won't work

---

Thomas A. Edison  
Kids, you tried your best and you failed miserably. The lesson is: "never try"

---

Homer Simpson

A primera vista,  **$\lambda$ Page** puede parecer simplemente un cuadro de texto al que se le agrega la posibilidad de interpretar expresiones *Haskell*. Esta visión es cierta, y en sí misma es un avance con respecto a las herramientas ya existentes pues permite intercalar en un mismo documento texto libre y expresiones *Haskell*.

Sin embargo,  **$\lambda$ Page** presenta varios atributos que generan un importante valor agregado:

- Permite configurar el entorno manual o automáticamente en base a un paquete *Cabal*
- Permite buscar documentación sobre expresiones *Haskell* utilizando *Hayoo!*
- Permite al usuario editar texto mientras espera el resultado de la evaluación de una expresión
- Permite visualizar y analizar expresiones que contengan errores, "bottoms" o que generen cálculos "infinitos" sin bloquearse ante su aparición y sin que ellos le impidan continuar presentando el resto de la expresión
- Permite cargar, importar y recargar módulos de modo de modificar el contexto de ejecución sin perder las expresiones con las que el usuario estaba trabajando
- Permite mantener varias páginas de expresiones por separado, guardarlas y reabrirlas de modo de organizar más amigablemente el entorno de trabajo del usuario

Son todas estas características las que convierten a  **$\lambda$ Page** en una herramienta de gran utilidad para todo desarrollador *Haskell*, desde el estudiante universitario que generalmente se encuentra frente a la necesidad de "entender" funciones del lenguaje y así aprenderlo hasta el desarrollador avanzado que necesita debuggear sus aplicaciones.

## 5.2. Trabajo a Realizar

Inside every large program, there is a small  
program trying to get out

---

C.A.R. Hoare

I'm a man with a one-track mind, so much to  
do in one life-time

---

Queen

**$\lambda$ Page** es todavía una aplicación en desarrollo y de código abierto. Aún queda mucho por hacer y por eso la hemos publicado en internet utilizando [github](#) [41]. Gracias a este servicio, se encuentra habilitada [una lista de bugs y sugerencias](#) en constante actualización. Entre las tareas que allí se encuentran al día de hoy, cabe destacar:

**Mejoras Visuales** La UI de  **$\lambda$ Page** tiene mucho por mejorar y optimizar. Ejemplos de esto son el coloreo de código y la autocompletación. *wxHaskell* se encuentra en constante desarrollo y sus optimizaciones deberían ser aprovechadas por  **$\lambda$ Page** cuando sea posible.

**Cabal** La actual integración con *Cabal* cubre lo básico de la configuración de paquetes, como las extensiones del lenguaje y los módulos, pero *Cabal* permite describir varias cosas más que podrían ser aprovechadas por  **$\lambda$ Page** como la ubicación de bibliotecas y otros archivos

**Auto Reload** Una característica que sería muy útil agregar a  **$\lambda$ Page** es la recarga automática de los módulos modificados. De este modo, el desarrollador no necesitaría presionar el botón “reload” cada vez que modifica un módulo para que  **$\lambda$ Page** tome los cambios realizados.

Por otra parte, si bien no menores, los progresos realizados por  **$\lambda$ Page** en cuanto a la presentación de interpretaciones de expresiones al usuario son sólo los primeros pasos en un camino en el cual todavía queda mucho por explorar. Por ejemplo:

**Visualizaciones** Actualmente,  **$\lambda$ Page** al interpretar expresiones puede mostrar distintos resultados dependiendo del tipo de la expresión. Sin embargo, sea cual fuese el resultado, no es otra cosa que texto. Esto es bastante completo pero hay situaciones en las que resultaría más conveniente otro modo de visualización. Podemos encontrar ejemplos de estos casos en los modelos de visualización que propone [Vital](#) [15], pero también, por ejemplo, para el caso de acciones de entrada/salida, sería útil contar con una consola que expusiese lo que estas acciones intentan mostrar por *stdout* o *stderr*.

**Tipos** En su afán de mostrar tanta información sobre las expresiones interpretadas como sea posible,  **$\lambda$ Page** trata de modo particular las listas. Pero las listas son sólo uno de los muchos tipos que pueden ser considerados a fin de presentar su interpretación de un modo particular. Podemos tomar como ejemplo el tipo **Maybe**, el tipo **Either**, los constructores de tuplas y, por que no, los tipos con constructores complejos que requieren varios valores de los cuales quizás sólo uno genera una excepción y el resto pueden ser interpretados sin problemas. Esto podría generalizarse creando una nueva clase, que permitiese definir, para cada tipo que la implementa, cómo deben ser evaluadas las expresiones de ese tipo a fin de obtener

toda la información posible sobre ellas más allá de que alguno(s) de sus componentes generen excepciones o cálculos “infinitos”.  **$\lambda$ Page** podría, luego, utilizar las funciones definidas por esa clase en caso de ser posible antes de recurrir a la función `show` al momento de interpretar una expresión.

Por otro lado,  **$\lambda$ Page** es sólo una de las múltiples herramientas comunes en el desarrollo de lenguajes orientados a objetos que pueden ser “migradas” al paradigma funcional.  **$\lambda$ Page** podría integrarse algún día en una IDE más completa y con mayores capacidades, que brinde al desarrollador funcionalidades tales como:

**Soporte para TDD** El ambiente de desarrollo de *Smalltalk* es un gran ejemplo de este tipo de herramientas ya que permite realizar (e incluso sugiere), al momento de detectar un test que falla, las tareas necesarias para hacerlo funcionar. Una herramienta de este tipo, combinada con el poder de QuickCheck [26] podría ser muy útil a la hora de desarrollar software *Haskell* utilizando Test Driven Development [1].

**Refactoring** [29] El proceso de mejorar el diseño de un programa sin alterar su comportamiento es una tarea de gran valor que puede ser automatizada por herramientas tales como **Wrangler** [28] para *Erlang*. Herramientas similares pueden ser incorporadas a una IDE para *Haskell* como la que aquí planteamos.

**Administración de Paquetes** Si bien  **$\lambda$ Page** permite trabajar con paquetes *Cabal*, no hace nada por su administración y mantenimiento. Una herramienta visual que permita crearlos, administrarlos y utilizarlos sería de gran ayuda. La extensión **EclipseFP** [31] para *Eclipse* brinda parte de esta funcionalidad.

**Análisis de Terminación** Dentro de  **$\lambda$ Page** o quizás dentro de esta IDE para *Haskell* que estamos planteando, podría tener cabida la herramienta **AProVE** [12] desarrollada por J. Giesl, S. Swiderski, P. Schneider-Kamp y R. Thiemann. Esta herramienta permite automatizar el chequeo de terminación de reescritura de términos para expresiones *Haskell*.

**Debugging** La tarea de “debuggear” código en el paradigma funcional es singularmente diferente a la misma en el paradigma imperativo. Sin embargo, una IDE completa para *Haskell* podría permitir ejecutar paso a paso las partes “imperativas” de las aplicaciones (o sea, aquellas que se encuentren dentro de la mónada **IO**).

Finalmente, debemos considerar que  **$\lambda$ Page** ha sido diseñado y desarrollado para *Haskell*, pero éste no es el único lenguaje que podría beneficiarse con una herramienta similar. Por ejemplo, sería interesante pensar en una “traducción” de  **$\lambda$ Page** a *Erlang*, considerando y **aprovechando** las diferencias entre ambos lenguajes.

## 6. Agradecimientos

Este proyecto nunca se podría haber llevado a cabo sin la ayuda de muchas personas que contribuyeron de una u otra manera a su realización. Quienes creamos  $\lambda$ **P**age queremos agradecer especialmente a aquellos que nos han ayudado en la comprensión, **corrección** y manejo de *wxHaskell*: [Arjan van IJzendoorn](#), [Eric Y. Kow](#) y [Jeremy O. Donoghue](#). También queremos agradecer a [Timo B. Hübel](#) y [Sebastian M. Schlat](#) que nos han permitido integrar  $\lambda$ **P**age con su herramienta *Hayoo!*. Y no podríamos dejar de mencionar a nuestros “beta-testers”: [Abram Hindle](#), [Mariano Perez Rodriguez](#), [Gustavo Salvini](#), [Facundo Villanueva](#), [Federico Grassi](#) y [Bernabé Panarello](#). Por último, pero no por eso menos importante, queremos agradecer a [Dario Ruellan](#) quien ha creado nuestra [página web](#).

Finalmente yo, [Fernando Benavides](#), quisiera agradecer a mi mujer, Constanza Zappala, que me ha acompañado, ayudado y soportado durante los casi dos años que duró el desarrollo de este proyecto, a Juan José Comellas, Alejandro Tolomei, Francisco de Ezcurra y toda la gente de [Novamens S.A.](#), la empresa en la que trabajo, por su interés y contribución al proyecto, a todos los profesores de *Algoritmos I* (Incluida la Caja Vengadora) y *Paradigmas de Lenguajes de Programación* por introducirme en el apasionante mundo de la programación funcional y especialmente en *Haskell*, y por supuesto, a mis profesores Daniel Gorín y Diego Garbervetsky, que desde el primer momento creyeron en mí y en esta idea loca de “hacer algo parecido a las cosas que tienen los que trabajan con objetos” con la que llegué a aquella primera reunión.

## Referencias

- [1] Kent Beck. Test Driven Development: By Example. Addison-Wesley Professional, 2002.
- [2] Fernando Benavides. eprocess [online]. October 2009. Available from: <http://hackage.haskell.org/package/eprocess>.
- [3] Fernando Benavides. λpage [online]. 2009. Available from: <http://haskell.hpage.com>.
- [4] Fernando Benavides. wxhnotepad [online]. December 2009. Available from: <http://github.com/elbrujohalcon/wxhnotepad>.
- [5] Fernando Benavides. hfiar [online]. January 2010. Available from: <http://hackage.haskell.org/package/hfiar>.
- [6] Fernando Benavides. Installing λpage from cabal [online]. May 2010. Available from: <http://wiki.github.com/elbrujohalcon/hPage/installing-page> [cited May 13, 2010].
- [7] Olaf Chitil et al. Hat - the haskell tracer [online]. Available from: <http://www.haskell.org/hat>.
- [8] Duncan Coutts and Don Stewart. Haskell platform [online]. Available from: <http://hackage.haskell.org/platform/>.
- [9] Paradigmas de Lenguajes de Programación. Práctica de Programación Funcional. Universidad de Buenos Aires, Segundo Cuatrimestre 2009. Available from: [http://www.dc.uba.ar/materias/plp/cursos/2009/cuat2/descargas/guias/Practica-funcional.pdf/at\\_download/file](http://www.dc.uba.ar/materias/plp/cursos/2009/cuat2/descargas/guias/Practica-funcional.pdf/at_download/file).
- [10] eclipse. Java scrapbook pages [online]. Available from: <http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.jdt.doc.user/reference/ref-34.htm>.
- [11] Ericsson. Erlang [online]. Available from: <http://www.erlang.org>.
- [12] J. Giesl, S. Swiderski, P. Schneider-Kamp, and R. Thiemann. Automated termination analysis for Haskell: From term rewriting to programming languages. Term Rewriting and Applications, pages 297–312, 2007.
- [13] Daniel Gorín. Hint [online]. January 2009. Available from: <http://projects.haskell.org/hint>.
- [14] James Gosling. Java [online]. October 2007. Available from: <http://www.java.com>.
- [15] Keith Hanna. A document-centered environment for haskell. Yet unpublished, April 2005. Available from: <http://www.cs.kent.ac.uk/projects/vital/description/2005/all.ps>.
- [16] Timo B. Hübel and Sebastian M. Schlat. Hayoo! [online]. Available from: <http://holumbus.fh-wedel.de/hayoo>.
- [17] Isaac Jones and Duncan Coutts. The haskell cabal [online]. Available from: <http://www.haskell.org/cabal>.
- [18] Mark P. Jones et al. Hugs [online]. Available from: <http://www.haskell.org/hugs/>.
- [19] Simon Peyton Jones. Ghci [online]. Available from: [http://www.haskell.org/ghc/docs/latest/html/users\\_guide/ghci.html](http://www.haskell.org/ghc/docs/latest/html/users_guide/ghci.html).
- [20] Simon Peyton Jones. Ghci debugger [online]. Available from: [http://www.haskell.org/ghc/docs/6.10-latest/html/users\\_guide/ghci-debugger.html](http://www.haskell.org/ghc/docs/6.10-latest/html/users_guide/ghci-debugger.html).

- [21] Simon Peyton Jones. The glasgow haskell compiler [online]. Available from: <http://www.haskell.org/ghc>.
- [22] Simon Peyton Jones, Paul Hudak, Philip Wadler, et al. Haskell [online]. Available from: <http://www.haskell.org>.
- [23] Damien Katz. Couchdb [online]. Available from: <http://couchdb.apache.org>.
- [24] Alan Kay, Dan Ingalls, and Adele Goldberg. Smalltalk [online]. Available from: <http://www.smalltalk.org>.
- [25] Alan Kay, Dan Ingalls, and Adele Goldberg. Smalltalk workspace [online]. Available from: <http://wiki.squeak.org/squeak/1934>.
- [26] John Hughes Koen Claessen. Quickcheck [online]. Available from: <http://www.cs.chalmers.se/~rjmh/QuickCheck/>.
- [27] Wilf R. LaLonde and John R. Pugh. Inside Smalltalk: vol. 1, volume I, page 71. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.
- [28] Huiqing Li. Wrangler [online]. 2008. Available from: <http://www.cs.kent.ac.uk/projects/forse/wrangler/doc/overview-summary.html>.
- [29] Huiqing Li. Refactoring Haskell Programs. Philosophy, University of Kent, September 2006.
- [30] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 333–343, New York, NY, USA, 1995. ACM. Available from: [http://portal.acm.org/ft\\_gateway.cfm?id=199528&type=pdf&coll=GUIDE&dl=GUIDE&CFID=91507314&CFTOKEN=66609812, doi:http://doi.acm.org/10.1145/199448.199528](http://portal.acm.org/ft_gateway.cfm?id=199528&type=pdf&coll=GUIDE&dl=GUIDE&CFID=91507314&CFTOKEN=66609812, doi:http://doi.acm.org/10.1145/199448.199528).
- [31] JP Moresmau, Leif Frenzel, and Thomas ten Cate. Eclipsefp [online]. Available from: <http://eclipsefp.sourceforge.net/>.
- [32] Jeremy O'Donoghue. Building a text editor. <http://wewantarock.wordpress.com>, January 2010.
- [33] Jeremy O'Donoghue and Eric Y. Kow. Wxhaskell [online]. Available from: <http://haskell.org/haskellwiki/WxHaskell>.
- [34] Bryan O'Sullivan, John Goerzen, and Don Stewart. Real World Haskell. O'Reilly Media, Inc., 1 edition, 2008.
- [35] Ross Paterson. Hackagedb [online]. Available from: <http://hackage.haskell.org>.
- [36] Chris Piro. Facebook chat [online]. February 2009. Available from: [http://www.facebook.com/note.php?note\\_id=51412338919](http://www.facebook.com/note.php?note_id=51412338919).
- [37] Dr. Robert Roebling, Dr. Vadim Zeitlin, Dr. Stefan Csomor, Dr. Julian Smart, Vaclav Slavik, Robin Dunn, et al. Wxwidgets [online]. Available from: <http://www.wxwidgets.org/>.
- [38] Alexey Shchepin. ejabberd [online]. Available from: <http://www.ejabberd.im>.
- [39] Ken Thompson. Bsd license definition [online]. 1969. Available from: <http://www.linbo.org/bsdlicense.html>.
- [40] Philip Wadler. Monads for functional programming. Advanced Functional Programming, pages 24–52, 1995.
- [41] Chris Wanstrath, PJ Hyett, and Tom Preston-Werner. Github [online]. Available from: <http://github.com>.