Tesis de Licenciatura

λPage

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

Directores

Dr. Diego Garbervetsky

Lic. Daniel Gorín

Abstract

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

Índice

1. Introducción

1.1. 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 [1], ejabberd [2] y el chat de Facebook [33], todos ellos desarrollados utilizando lenguajes del paradigma funcional.

Ejemplos de éstos lenguajes de programación, como Haskell [9] o Erlang [3], demuestran ser maduros, confiables y presentan claras ventajas en comparación con los lenguajes tradicionales del paradigma imperativo. 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"*[1] .

Los desarrolladores Haskell cuentan actualmente con dos herramientas de este tipo:

GHCi **[4]** La consola que provee GHC [7] permite a los desarrolladores evaluar expresiones, verificar su tipo o su clase. Cuenta también con un mecanismo de debugging [5] 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

Hat **[12]** Un 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 quizá 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.

[1] Entiéndase "micro-testing" como la tarea de realizar tests eventuales para entender o evaluar algún aspecto de un programa

1.2. 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 [14] para Java [20] y Workspace [17,30] para SmallTalk [16]. 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 guardardas 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 [28]. *Vital* es una implementación de un entorno de visualización de documentos para *Haskell*. Pretende presentar *Haskell* de una manera apropiada para usuarios finales en areas 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 **λ*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 [24] 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 [31] en sucesivos artículos en su blog

1.3. λ*Page*

> 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 brieve 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

λ***Page*** [23] 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. λ***Page*** es capaz de identificar las expresiones y definiciones válidas y permite al desarrollador inspeccionarlas, evaluarlas, conocer su tipo y su clase.

En el espíritu de las herramientas provistas por la comunidad de desarrolladores *Haskell*, λ***Page*** se integra con *Cabal* [10] y Hayoo! [13] y se encuentra ya disponible en HackageDB [8].

λ***Page*** presenta una interfaz simple e intuitiva, desarrollada utilizando wxHaskell [18], lo que lo convierte en una aplicación multiplataforma.

Por ser una herramienta desarrollada con *Haskell* para *Haskell*, λ***Page*** se diferencia de sus pares del mundo de objetos, al aprovechar conceptos claves como son el tipado fuerte (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", λ***Page*** 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.

λ***Page*** se encuentra desarrollado utilizando *eprocess* [22], una librería que facilita el manejo de "threads" en un estilo similar al de los procesos *Erlang*. Gracias al uso de esta librería, λ***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. También gracias al uso de *eprocess*, λ***Page*** permite detectar cálculos infinitos (o más

precisamente, cálculos que demoran demasiado) e informar sobre este hecho al usuario para que ya no siga esperando indefinidamente el resultado de la evaluación solicitada.

2. Descubriendo λ*Page*

2.1. 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 λ***Page*** en *OSX* o *Windows*, se proveen instaladores en el sitio web de λ***Page***, sin embargo,como se ha dicho, λ***Page*** 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+ **[19]** El framework de desarrollo para interfaces de usuario que utiliza *wxHaskell*. Debe ser instalado con los módulos unicode, cmdline, config, log, stl, richtext y clipboard, al menos y con el módulo odbc desactivado.

Haskell Platform **[11]** Una distribución de *Haskell* que incluye todo lo necesario para compilar e instalar programas desarrollados en este lenguaje (de particular interés para λ***Page***: GHC y happy).

2.1.1. Windows

Para el correcto funcionamiento de λ***Page*** los usuarios de *Windows XP* deben instalar el C++ 2008 SP1 [15].

2.1.2. Linux

En algunas distribuciones de Linux es conveniente, además de la instalación de *Haskell Platform* instalar las librerías de *Monad Transformers* ejecutando, por ejemplo:

```
$ sudo aptitude install libghc6-mtl-dev libghc6-mtl-doc
```

2.2. Caso de Uso: Aprobando PLP con *λPage*

> Es como un árbol, pero al revés: con la raíz abajo
>
> Eduardo Bonelli, profesor de PLP

> 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 Simpons

Mostraremos a continuación, a través de un ejemplo, cómo utilizar *λPage*. En este caso, hemos tomado prestada una práctica de la materia *Paradigmas de Lenguajes de Programación* [26]. Exhibiremos entonces, cómo un alumno podría utilizar *λPage* para resolver algunos de los ejercicios que allí se presentan. Seleccionaremos sólo aquellos que a nuestro criterio son los más representativos a la hora de entender cómo *λPage* ayuda al alumno en su resolución.

2.2.1. Pasos previos

Antes de comenzar a resolver los ejercicios, el alumno ejecuta *λPage* y clickea en el botón *New*. El programa presentará una pantalla similar a la de la figura 1.

2.2.2. 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

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

Resolución Para resolver este ejercicio, el alumno debe reformular el código provisto por la cátedra para que *λPage* lo interprete correctamente. Para ello debe "encerrar" entre paréntesis la primera expresión y quitar el nombre de la segunda de modo que el resultado obtenido al presionar el botón *Interpret* corresponda al valor de `ys`. Además debe insertar un renglón en blanco luego de la definición de `xs` para que *λPage* pueda observar que se trata de dos expresiones diferentes. Luego de ello, simplemente presiona el botón *Interpret* y puede observar el tipo del resultado (`[Float ->Float]`), tal como lo muestra la figura 2.

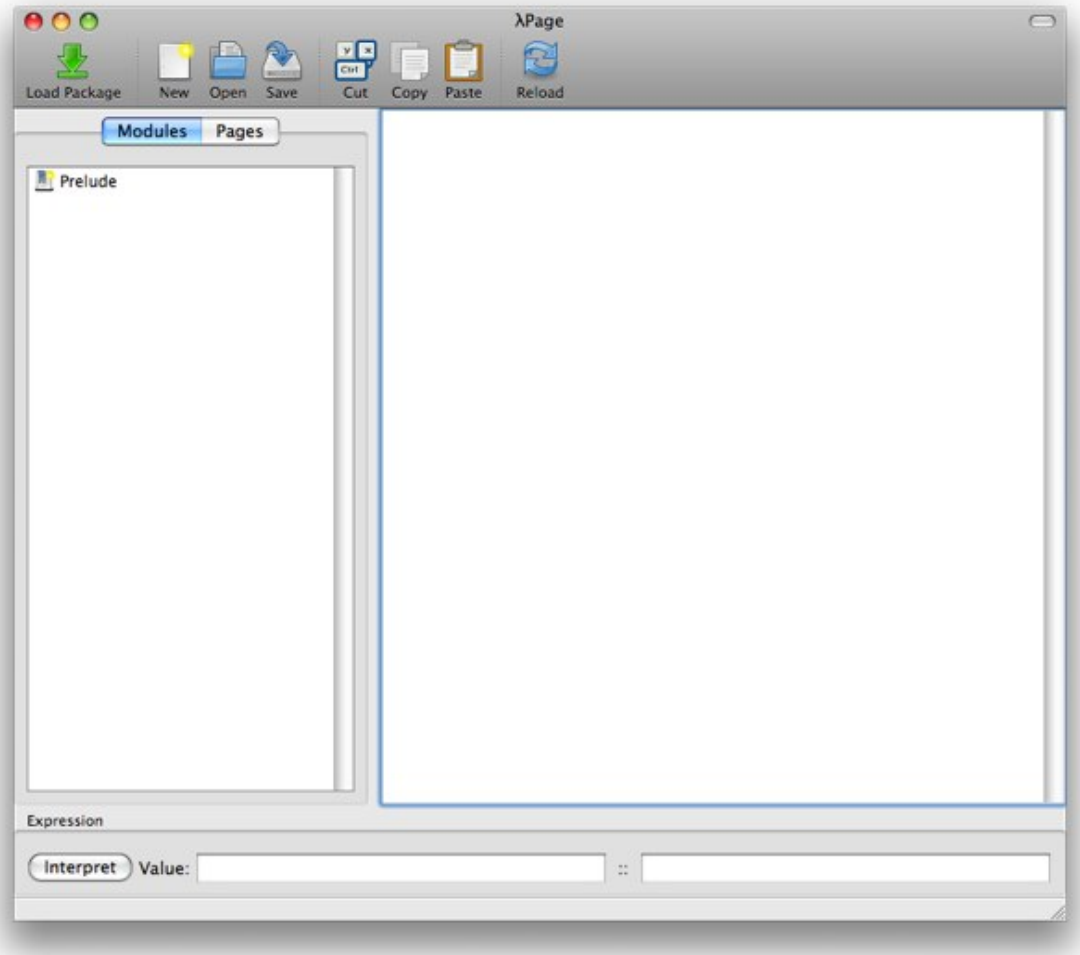

Figura 1: Tutorial - Previo a comenzar

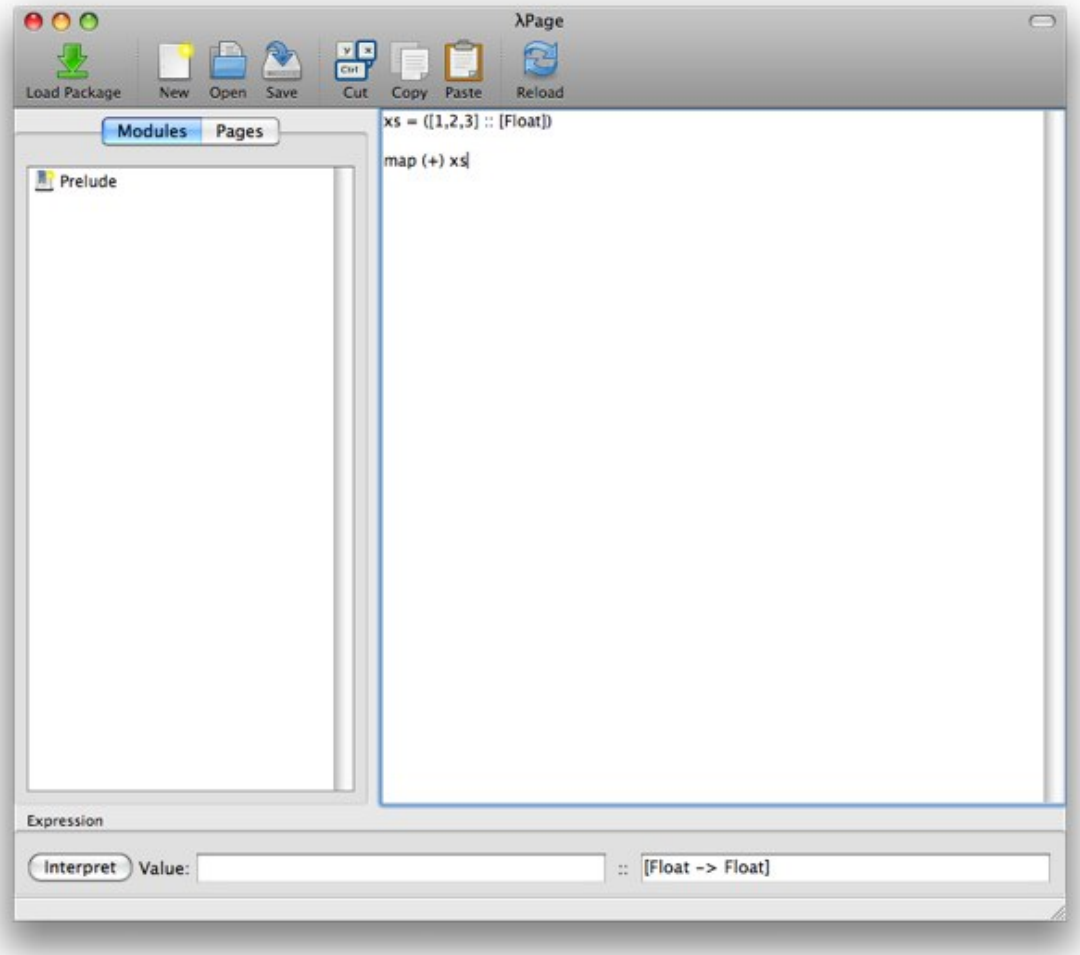

Figura 2: Tutorial - Ejercicio 1

2.2.3. Listas por Comprensión

Ejercicio 4 ¿Cuál es el valor de esta expresión?

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

Resolución Este ejercicio se resuelve simplemente copiando y pegando el código dentro de ***λPage***, seleccionando la expresión a evaluar (si es que no se ha borrado las expresiones anteriores) y evaluándola con el botón *Interpret*. ***λPage*** mostrará entonces el resultado: `[1,1,1,2,2,3,3,4]` como se ve en la figura 3.

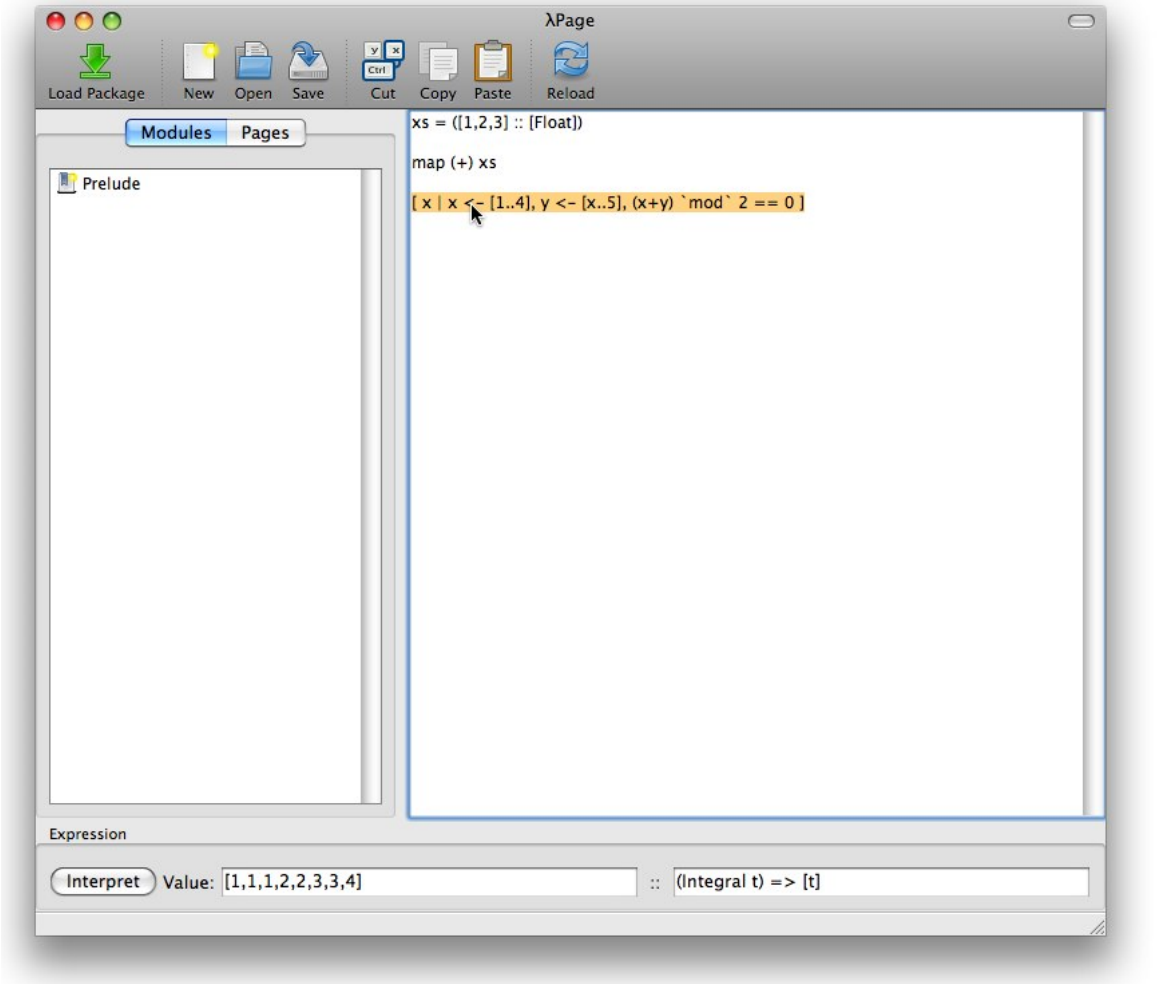

Figura 3: Tutorial - Ejercicio 4

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 triplas 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]
```

Resolución Para resolver este ejercicio, el alumno podría comenzar por intentar evaluar la lista que se le provee, para ello reacomodará su definición tal como se observa en la figura 4 y presionará el botón *Interpret*.

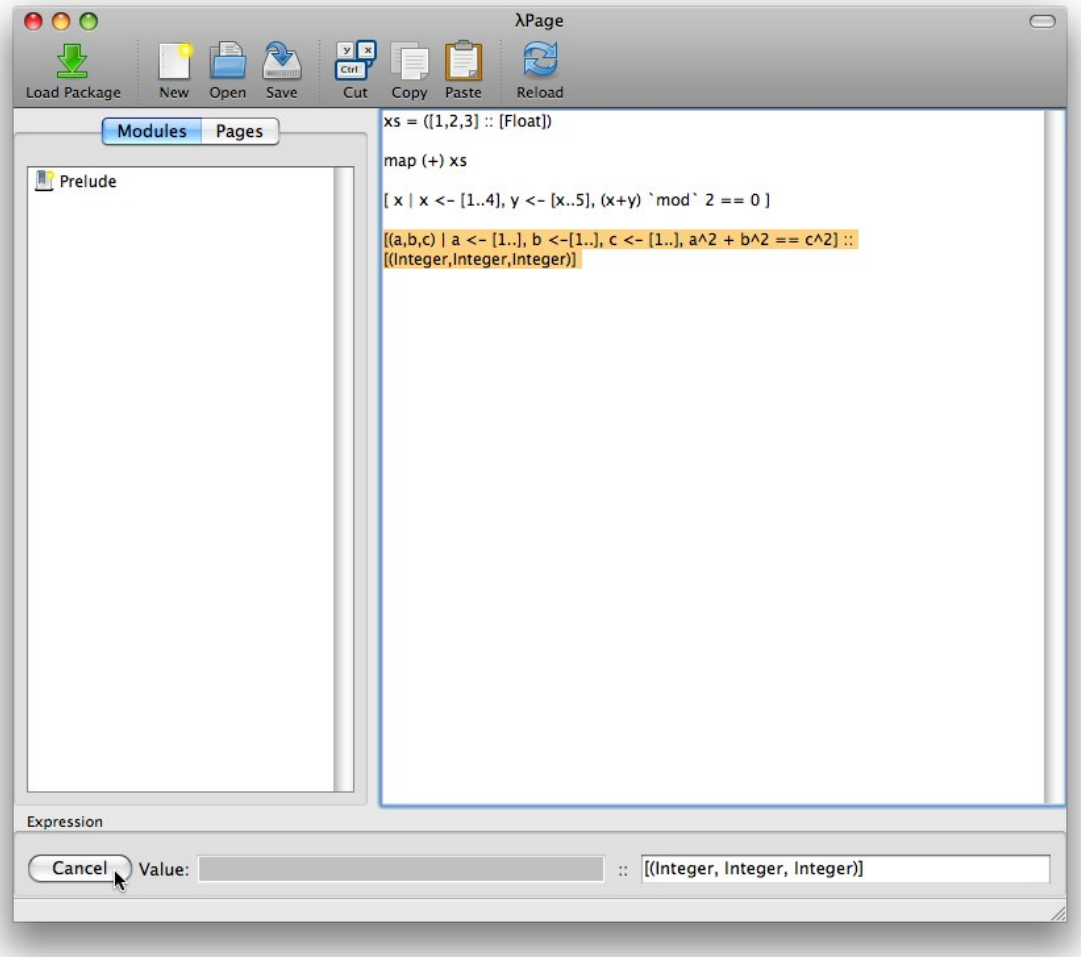

Figura 4: Tutorial - Ejercicio 5 - Primer intento

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`), para generar el primer elemento de la lista, *Haskell* 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, *Haskell* nunca encontrará el primer elemento de esta lista. **λPage** 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 **λPage** trata de evaluar la expresión, el alumno puede modificar la expresión 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 $b > c \Rightarrow b^2 > c^2$. **λPage** entonces, comenzará a exhibir resultados hasta que el alumno presione el botón *Cancel*.

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*.

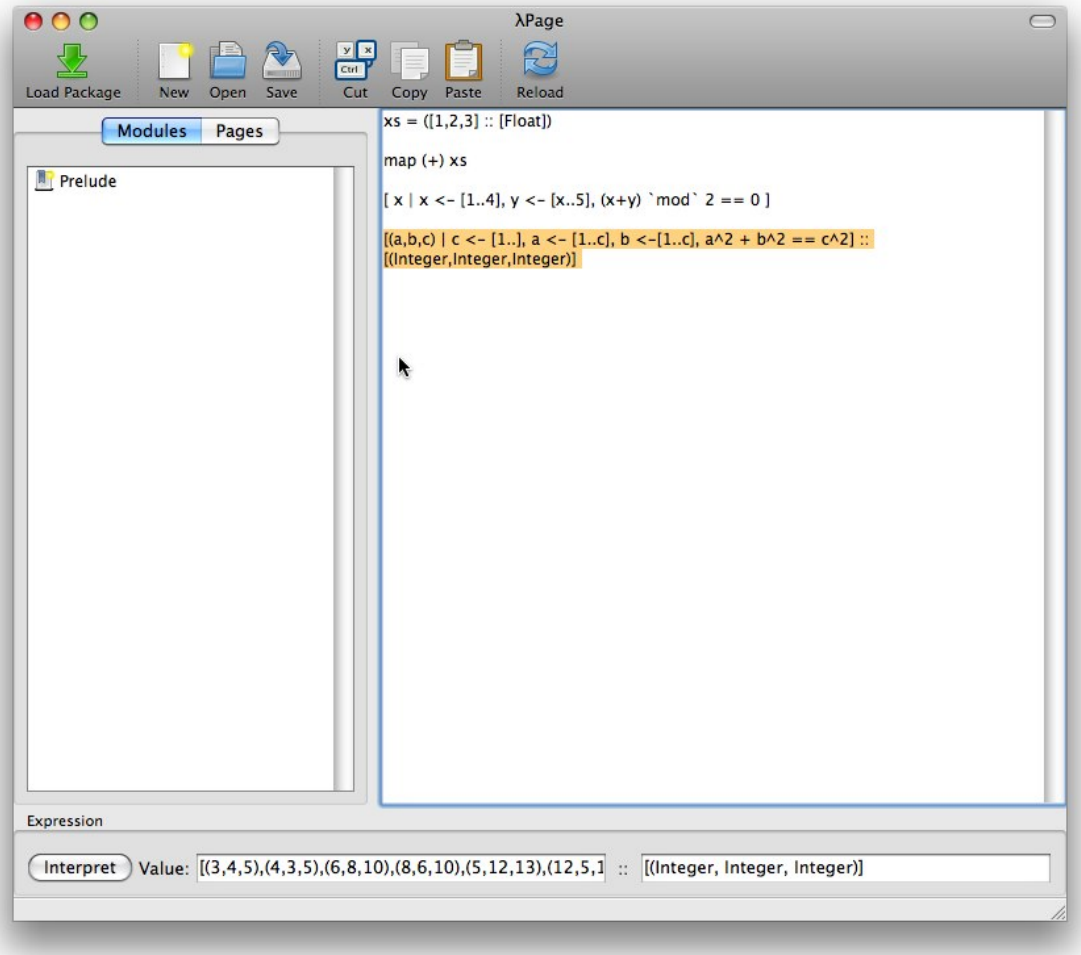

Figura 5: Tutorial - Ejercicio 5 - Segundo intento

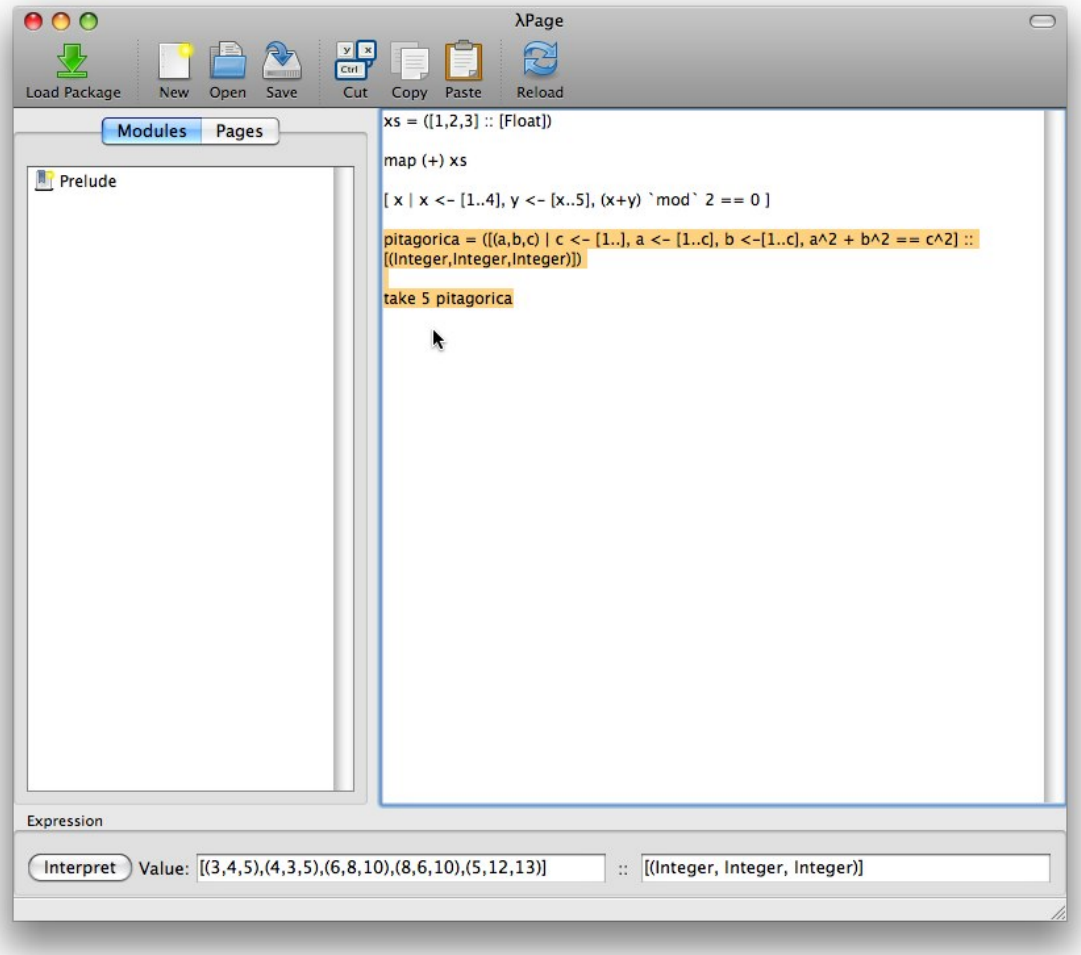

Figura 6: Tutorial - Ejercicio 5 - Tercer intento

2.3. Caso de Uso: Ganando al 4 en Línea con λ*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

Incluímos en este informe un segundo tutorial, apuntando esta vez a demostrar cómo λ***Page*** puede ser de utilidad para un programador *Haskell* que se enfrenta a un problema más lejano al mundo académico. En él veremos cómo trabajar con herramientas tales como *Cabal*, *Hayoo!* y *wxhaskell*. Veremos como λ***Page*** ayuda al usuario a "entender" código escrito por otra persona (o quizá por el mismo, suficiente tiempo atrás).

2.3.1. Introducción

La historia comienza cuando a nuestra amiga desarrolladora, a quien llamaremos *Fátima* para darle un poco de personalidad, se encuentra con la misión de modificar una implementación de un juego de *4 en Línea*, llamada hfiar [25]. 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 λ***Page***.

2.3.2. 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.

Resolución 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.

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.

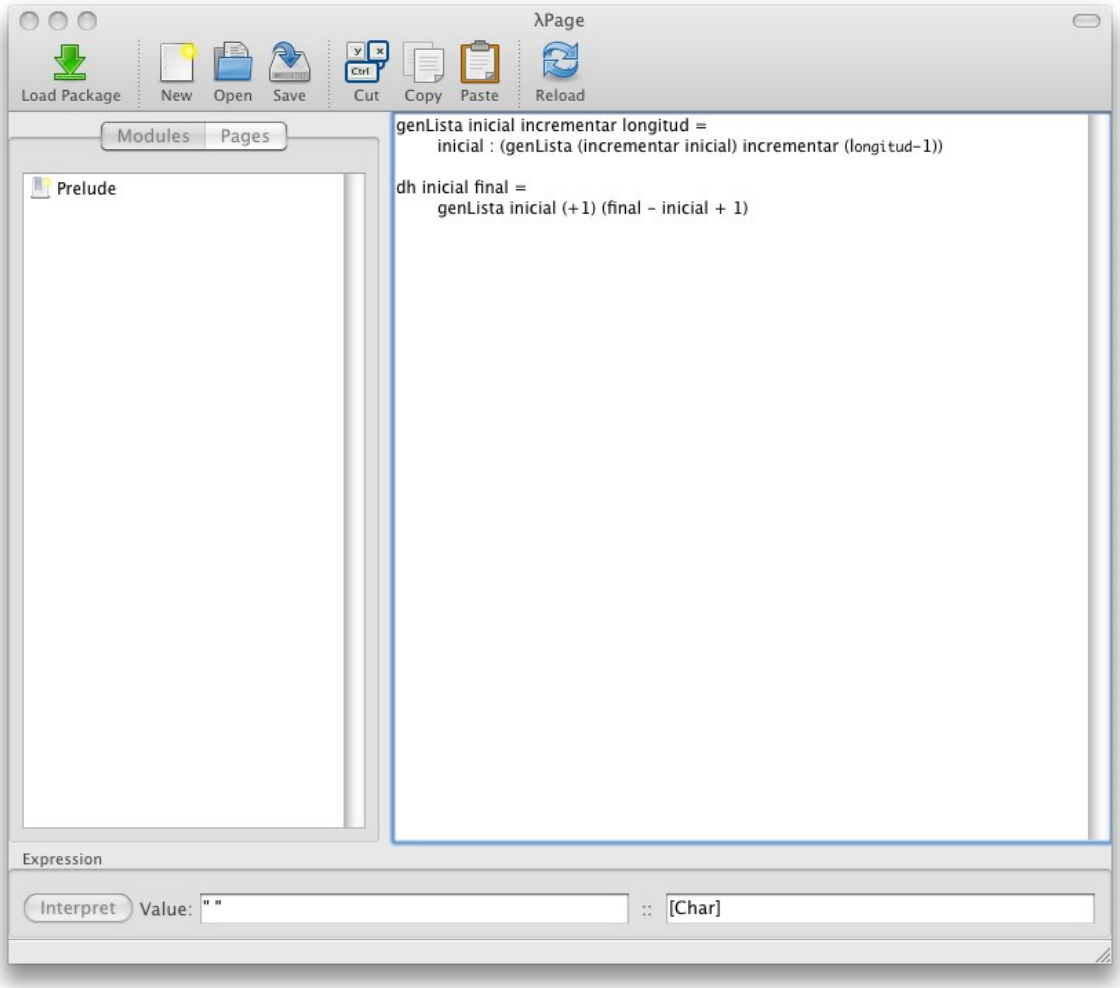

Figura 7: Tutorial - Ejercicio 9 - Primer Intento

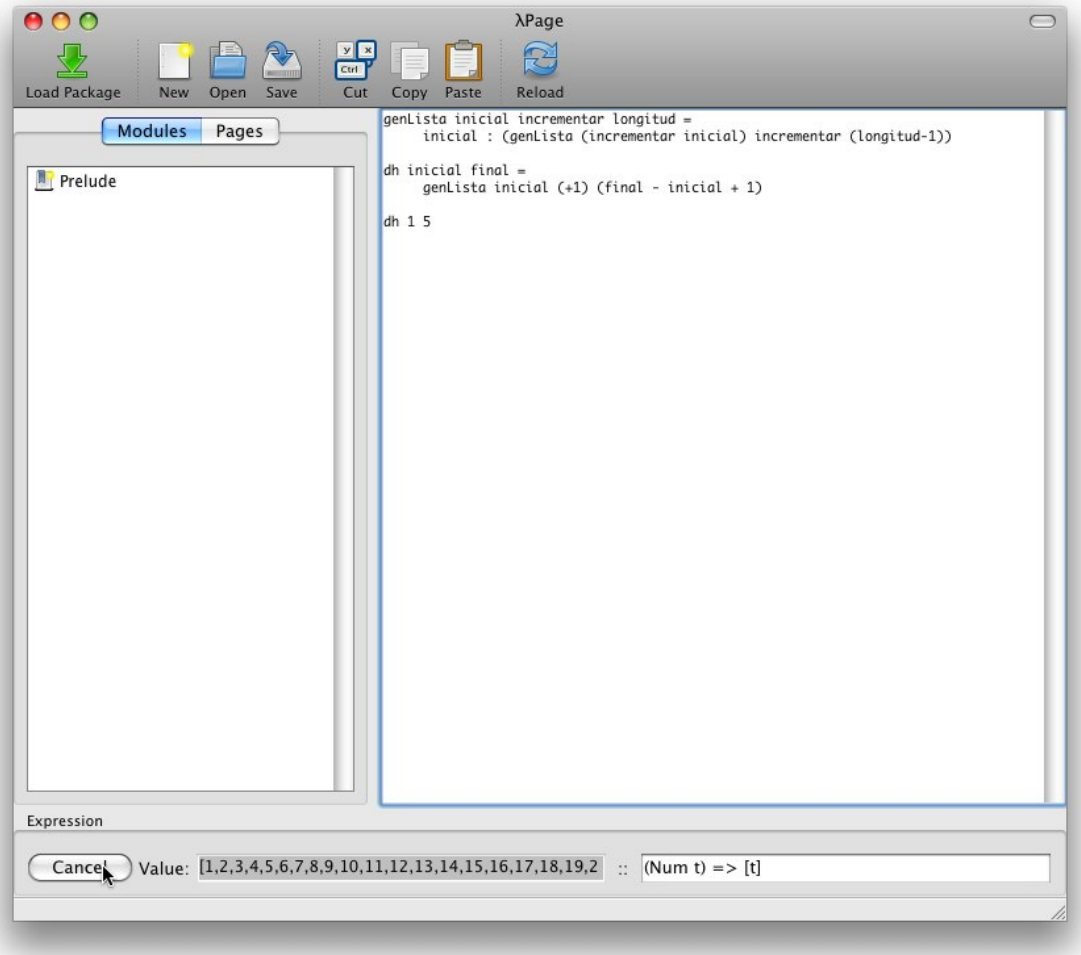

Figura 8: Tutorial - Ejercicio 9 - Recursión Infinita

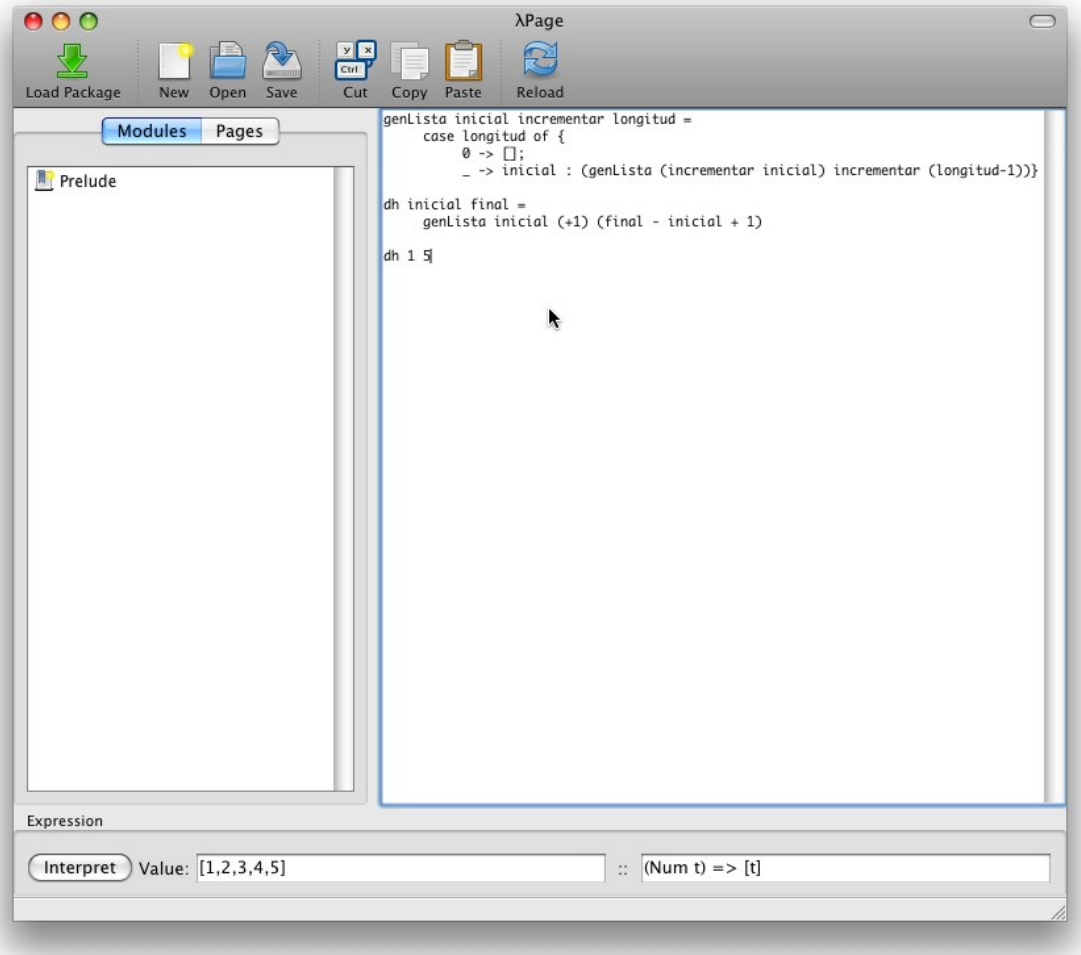

Figura 9: Tutorial - Ejercicio 9 - Segundo Intento

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))
```

...

Resolución El ejercicio continúa, pero en este caso, el alumno podría verse tentado a intentar evaluar `miAgenda` directamente en ***λPage*** y obtendría el resultado de la figura 10. Ésto se debe a que ***λPage*** no soporta definiciones de tipos de datos directamente en el texto.

Para conseguir el efecto deseado, el alumno puede crear un módulo (sin salir de ***λPage***) y guardarlo utilizando la opción *File* → *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 cláusla `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 ***λPage*** permite hacer con los elementos de la lista *Modules*. Presionando botón derecho del mouse sobre un ítem, ***λPage*** despliega un menú 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`

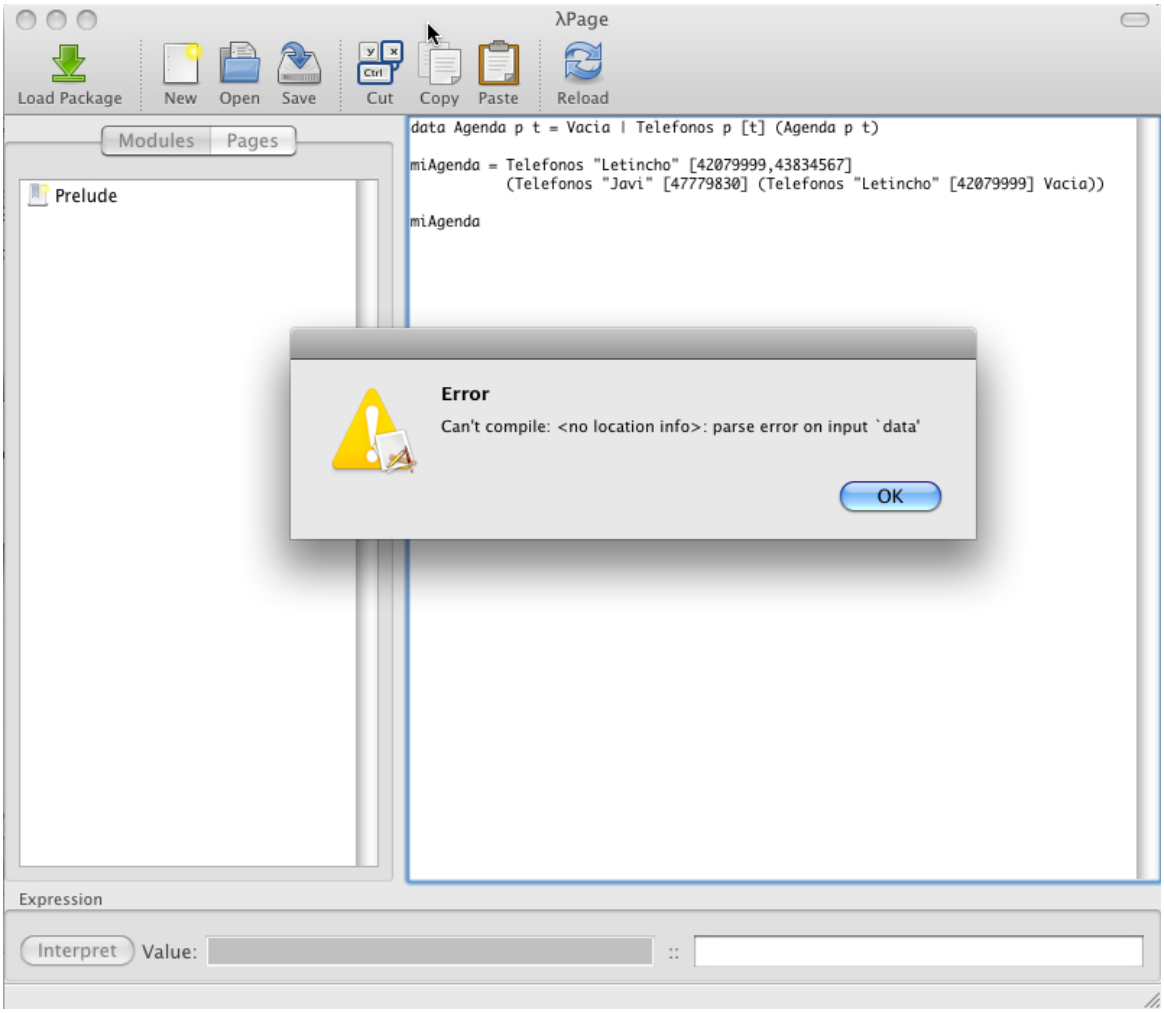

Figura 10: Tutorial - Ejercicio 23 - Primer Intento

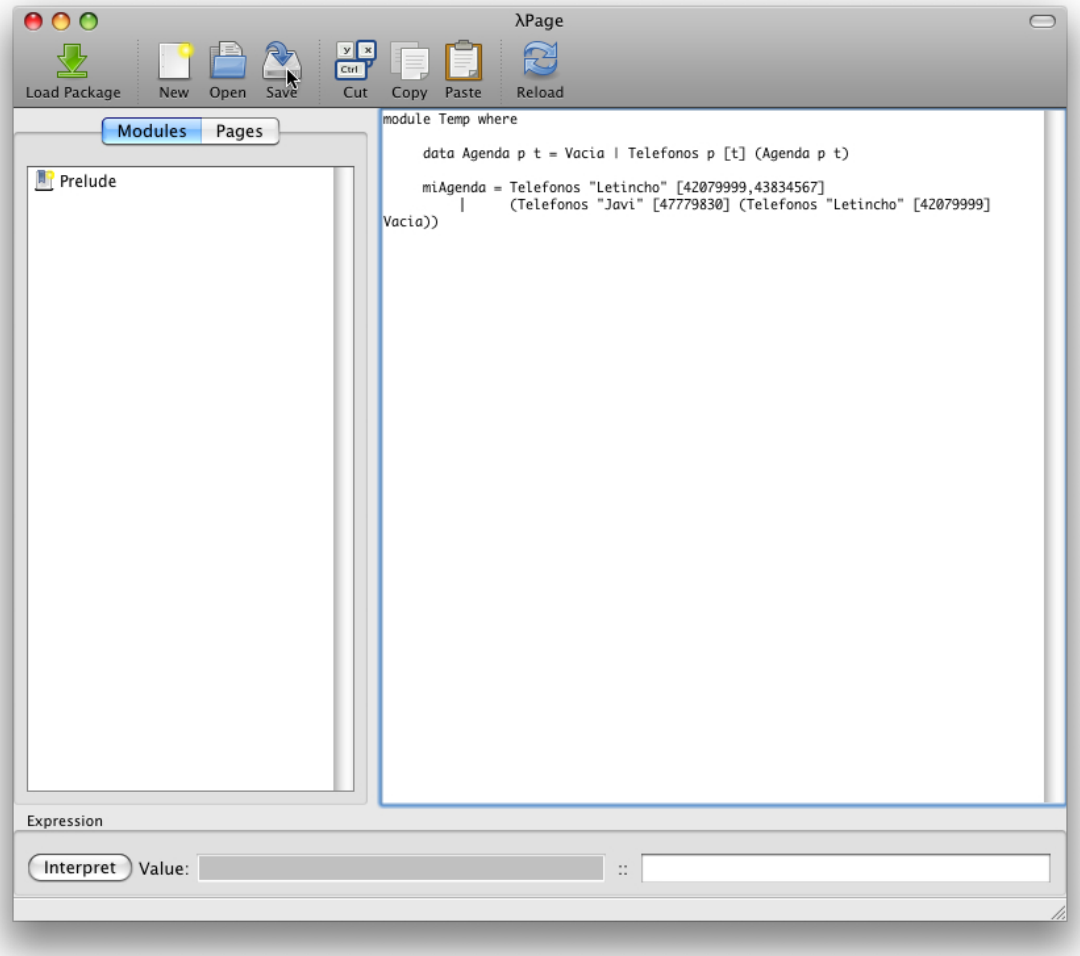

Figura 11: Tutorial - Ejercicio 23 - Crear Módulo

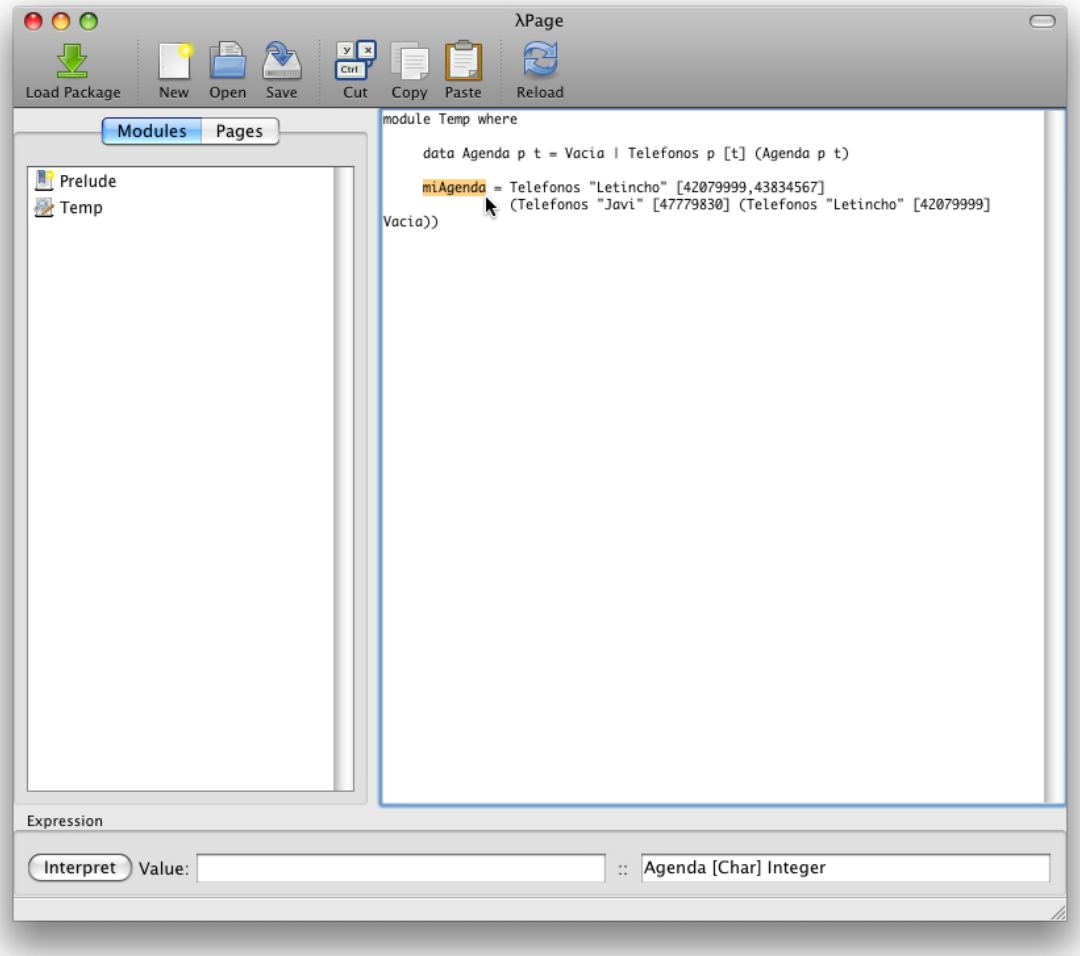

Figura 12: Tutorial - Ejercicio 23 - Segundo Intento

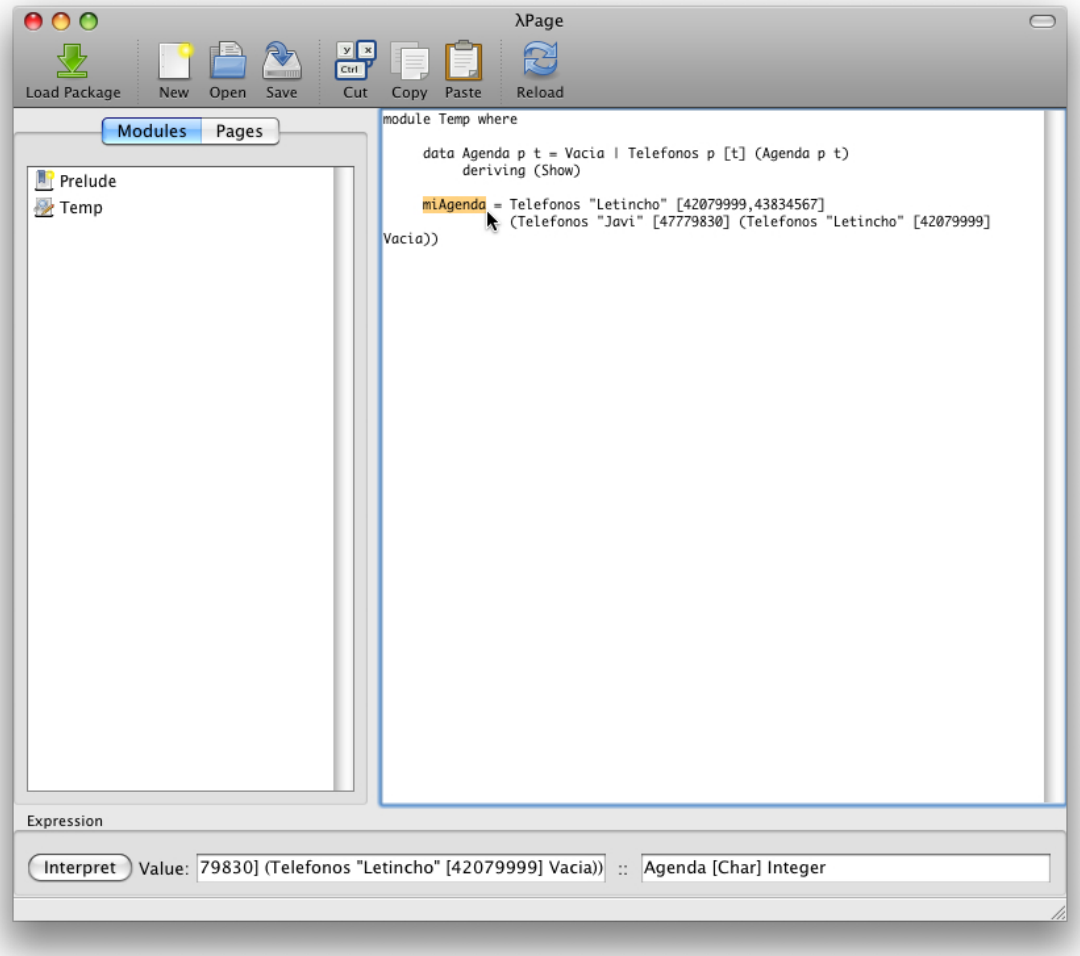

Figura 13: Tutorial - Ejercicio 23 - Tercer Intento

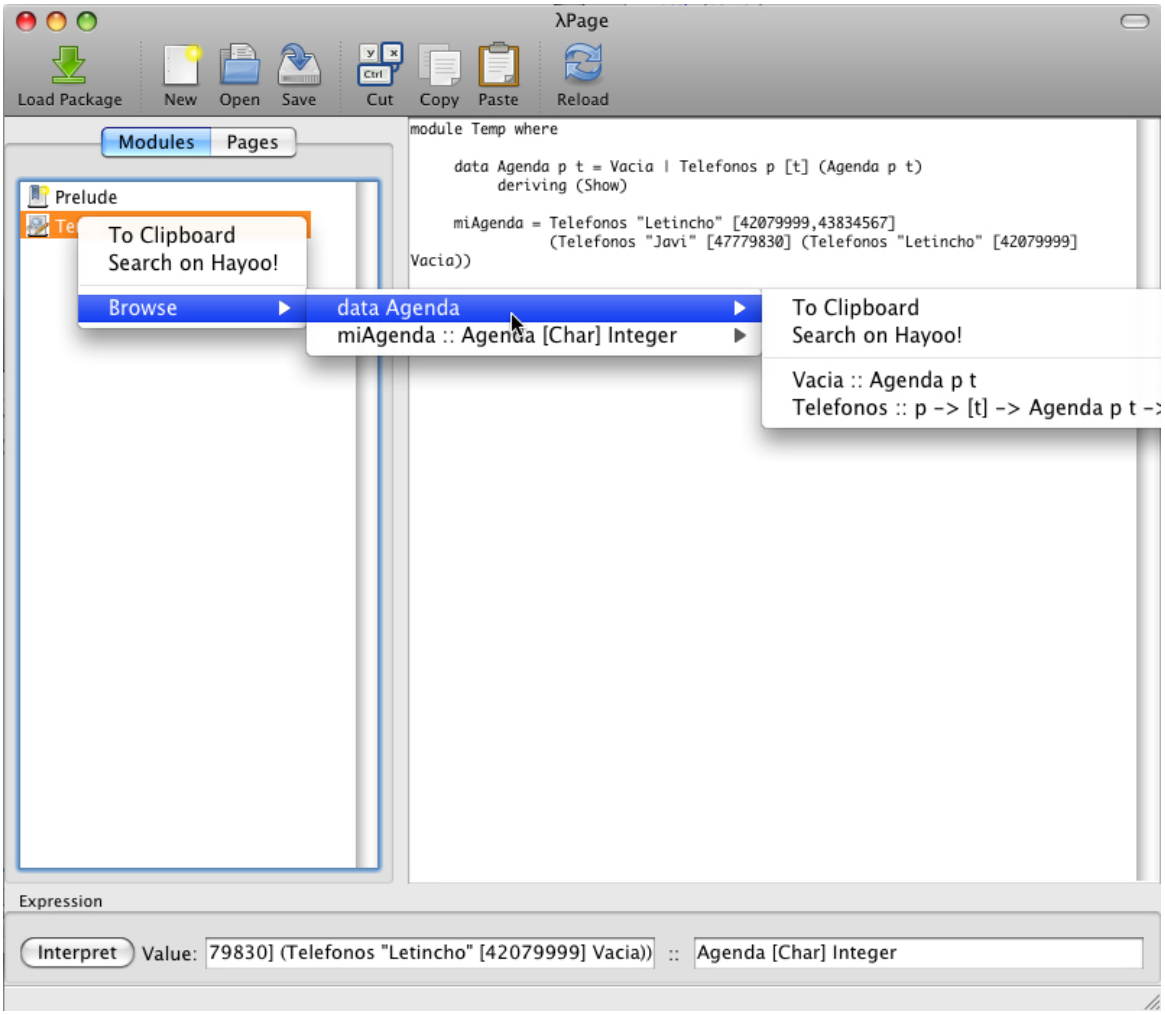

Figura 14: Tutorial - Navegando Módulos

2.3.3. Conclusiones

En este tutorial hemos demostrado un uso sencillo de **λPage** 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 **λPage** permite definir expresiones y funciones, para luego evaluarlas de manera individual o combinada. Dichas evaluaciones pueden generar diversos resultados y hemos visto cómo se maneja **λPage** con algunos de ellos:

- Para aquellas expresiones cuyo valor no puede ser expresado en forma de texto, hemos visto cómo **λPage** nos permite conocer su tipo.
- En el caso de expresiones cuyo valor es de longitud infinita, **λPage** exhibe todo lo que el usuario desee del resultado, culminando cuando éste presiona el botón *Cancel*.
- Y en relación a aquellas que requieren un cálculo infinito para determinar su valor, **λPage** 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 **λPage** de editar definiciones de manera simple y directa, para luego volver a evaluar expresiones que las usan, permite al usuario trabajar de fluidamente y le da la libertad de *cometer errores*, detectarlos, corregirlos y luego continuar su trabajo. Esta característica de **λPage** 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 **λPage** para crear, cargar, modificar y recargar módulos, trabajando siempre en una única página de texto. Ésta es otra característica de **λPage** que facilita el trabajo ya no sólamente a los estudiantes sino a todos los tipos de desarrolladores *Haskell*.

3. Desarrollo - ¿Cómo se hizo *λPage*?

3.1. 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 ***λPage*** tuvieron como principales motivaciones los siguientes requerimientos:

Conexión con GHC ***λPage*** 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 [27]

Paralelismo ***λPage*** debía permitir al usuario editar sus documentos mientras esperaba el resultado de la evaluación de alguna expresión. Para ello se creó *eprocess* y se implementó un modelo de procesos utilizándolo.

Errores Controlados ***λPage*** no debía fallar si la evaluación de una expresión fallaba. Más aún, también debía detectar posibles evaluaciones infinitas e informar estas situaciones al usuario

Teniendo en cuenta estos requerimientos, la arquitectura resultante puede ser descripta con el diagrama de la figura 15. Esta figura presenta el estado del sistema en un instante dado. Cada bloque representan un proceso o "thread" en ejecución. 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**, captura sus errores y lo reinicia en caso de ser necesario. En este proceso se mantiene el estado general de la aplicación: sus páginas, expresiones, paquetes y módulos cargados, etc.

Hint Server Este proceso, iniciado por el **HPage Server**, mantiene una conexión con la máquina virtual de GHC (a la cual se muestra en la figura conectado a través de una línea de puntos)

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 caracter (o sea, una expresión de tipo Char), para luego evaluarlo y enviar como respuesta su valor en forma normal.

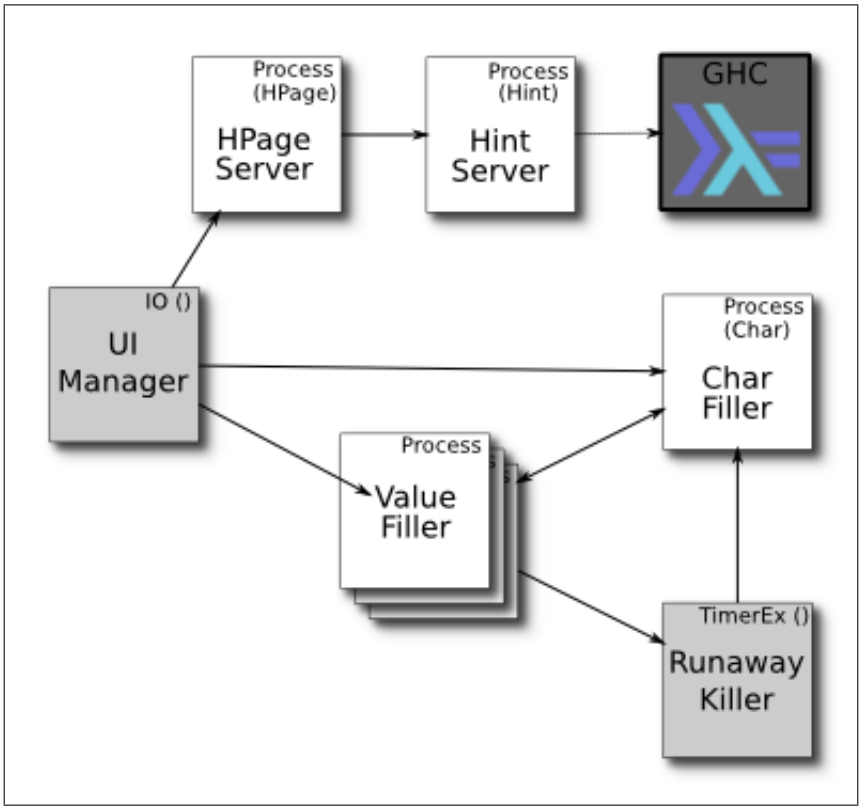

Figura 15: Arquitectura de λ***Page***

Value Filler Estos procesos, iniciados por el **UI Manager** ante cada evaluación solicitada por el usuario son los encargados 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. Cada **Value Filler** se encarga de evaluar un resultado y mostrarlo por pantalla, para ello envía y recibe mensajes del **Char Filler** a fin de procesar cada caracter a mostrar.

Runaway Killer Este thread, creado utilizando la clase *TimerEx* provista por *wxHaskell*, es iniciado por cada **Value Filler** al momento de enviar un nuevo caracter al **Char Filler**. El objetivo del **Runaway Killer** es el de detectar prorcesamiento "posiblemente" infinito. Básicamente, pasado un segundo de procesamiento, reinicia el **Char Filler** e informa al **Value Filler** que lo inició que el caracter que se esperaba procesar ha demorado demasiado y podría desencadenar una evaluación infinita.

Para un mayor detalle, la figura 16 nos muestra un diagrama de secuencia correspondiente a un proceso de evaluación. Para poder brindar un ejemplo completo, hemos "fabricado" un tipo que responde al siguiente código:

```
data WithIfiniteChar = WIC

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

Como puede observarse, al intentar mostrar la expresión `WIC`, ***λPage*** se encontrará con una cadena cuyo segundo caracter no puede computar pues requiere un cálculo infinito, en la figura representamos este caracter con la letra Ω. Allí es donde entra en acción el **Runaway Killer** para informar esta situación al usuario.

3.2. 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 ***λPage***. 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!*.

3.2.1. Concurrencia

Como hemos visto en la sección anterior, al momento de diseñar ***λPage*** tuvimos que considerar la necesidad de paralelizar tareas, para permitir al usuario, por ejemplo, trabajar en un documento mientras el motor de ***λPage*** 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

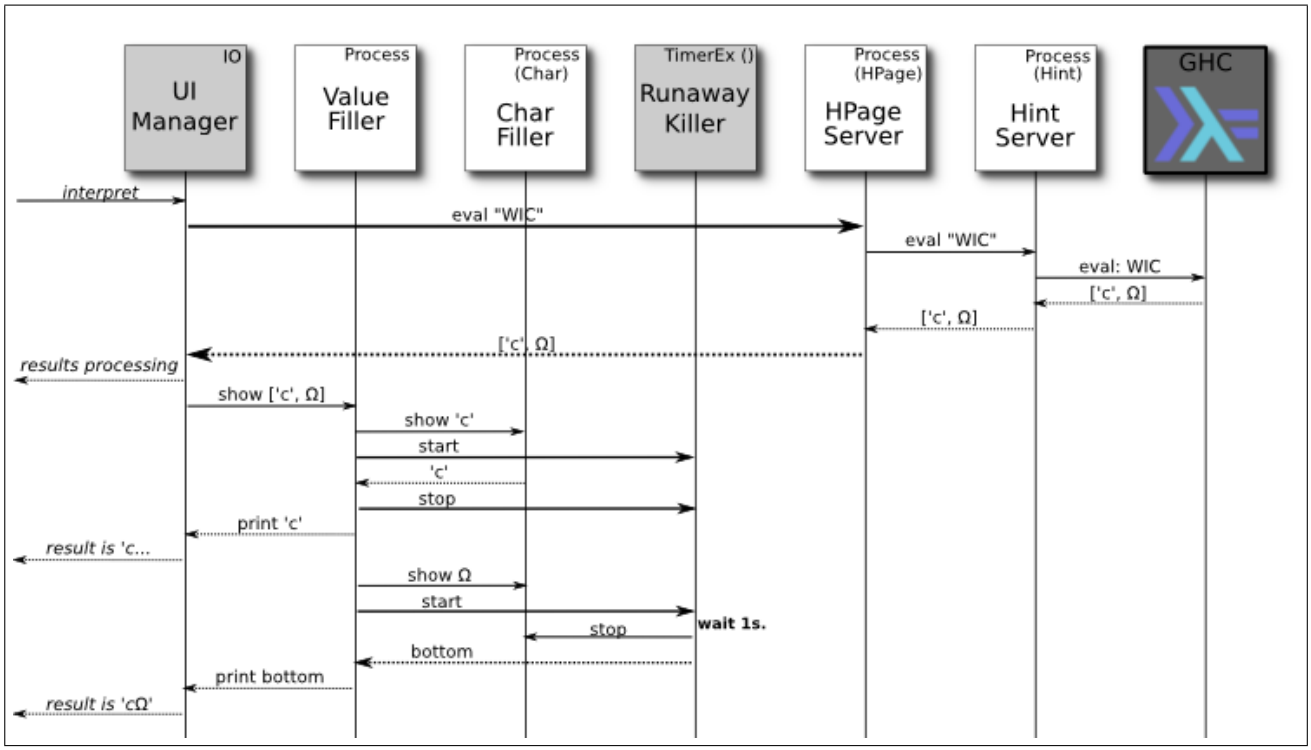

Figura 16: Secuencia de Evaluación de “WIC”

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 uso de mónadas, 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 (**HPage Server** y **Hint Server**), como veremos luego en la sección *Implementación*.

3.2.2. 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 **λPage** 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 **λPage** las expresiones son reducidas a forma normal al momento de intentar mostrar el resultado de su evaluación al usuario. En ese momento, lo único que se sabe es que la expresión a mostrar es de clase *Show*, lo que quiere decir que la misma es de tipo `[Char]`. **λPage** intenta entonces 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 caracter, se genere un cálculo "infinito" o una excepción. En este caso, **λPage** aborta el proceso que se encuentra intentando evaluar el caracter en cuestión e informa este hecho al usuario, para luego continuar evaluando los siguientes caracteres
- Otra posibilidad es que, luego de presentar un caracter, al intentar obtener el resto de la cadena, **λPage** encuentre un cálculo "infinito" o una excepción. En estos casos, se informa la situación al usuario y se aborta la evaluación de la expresión.

3.2.3. Integración

Una de las herramientas más comunmente usada por los desarrolladores haskell es *Cabal*. *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*. En un *paquete Cabal*, el desarrollador define los módulos que componen su aplicación o librería, los lugares (carpetas) donde 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!*. *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 *Haskell*. 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.

3.3. 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

3.3.1. eprocess

Para la implementación de *eprocess*, nuestra librería de procesos, utilizamos dos herramientas de paralelismo y concurrencia que se encuentran muy bien descriptas en el libro Real World Haskell [32]. Utilizamos *Threads* para paralelizar procesos y *Channels* y *MVars* para permitirles comunicarse. 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 dentienen un proceso respectivamente. `spawn` retorna como resultado un `Handle`, que identifica unívocamente 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*.

3.3.2. 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 *λPage* (correspondientes a la mónada `HPage`) de las relativas a la interfaz de usuario, para lo cual creamos el proceso **HPage Server**. Gracias al uso de mónadas y los mecanismos provistos por *eprocess*, pudimos construir éstos dos servers de una manera sencilla. Mostramos, a modo de ejemplo e incluyendo comentarios explicativos, el código más significativo de uno de ellos:

```
-- Para comenzar definimos un tipo ``ServerHandle'', que debe ser utilizado para
-- enviarle mensajes al servidor
newtype ServerHandle = SH {handle :: Handle (InterpreterT IO ())}

-- Convertimos a (ReceiverT r m) en una instancia de MonadInterpreter para poder
-- ejecutar las acciones que esperamos recibir como mensajes
instance MonadInterpreter m => MonadInterpreter (ReceiverT r m) where
    fromSession = lift . fromSession
    modifySessionRef a = lift . (modifySessionRef a)
    runGhc = lift . runGhc

-- Definimos una funcion que inicia el servidor y retorna su handle
-- El servidor simplemente recibe acciones de tipo MonadInterpreter
-- y las ejecuta utilizando la funcion lift
start :: IO ServerHandle
start = (spawn $ makeProcess runInterpreter interpreter) >>= return . SH
    where interpreter =
```

```
            do
                setImports ["Prelude"]
                forever $ recv >>= lift

-- Definimos una funcion para ejecutar acciones asincronicamente
-- Esta funcion recibe la acciona a ejecutar junto con el handle del servidor
-- y devuelve una MVar que llenara con el resultado de la accion
asyncRunIn :: ServerHandle -> InterpreterT IO a
    -> IO (MVar (Either InterpreterError a))
asyncRunIn server action = do
    resVar <- liftIO newEmptyMVar
    sendTo (handle server) $ try action >>= liftIO . putMVar resVar
    return resVar

-- Definimos tambien una funcion para ejecutar resultados sincronicamente
-- Esta funcion envia al servidor la acciona a ejecutar y espera recibir el
-- resultado para luego devolverlo
runIn :: ServerHandle -> InterpreterT IO a
      -> IO (Either InterpreterError a)
runIn server action = runHere $ do
    me <- self
    sendTo (handle server) $ try action >>= sendTo me
    recv

-- Finalmente, definimos una funcion que detiene al servidor utilizando la
-- la funcion kill
stop :: ServerHandle -> IO ()
stop = kill . handle
```

Cabe destacar la simpleza de este módulo que, con no más de 25 líneas de código es capaz de ejecutar todas las acciones que provee *hint* en un proceso aislado y controlado.

3.3.3. Módulos de *λPage*

El módulo principal de la aplicación es el denominado `HPage.Control`. Este módulo describe la mónada `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.

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

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

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

```
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,
                         -- Actions --
                         running :: Maybe InFlightData,
                         recoveryLog :: Hint.InterpreterT IO ()
```

```
                    }
```

Notese 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, como 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 eventual caída del **Hint Server**, el **HPage Server** lo reinicia y restaura su estado utilizando esos datos.

Finalmente, `running` y `recoveryLog` se utilizan para permitir al usuario cancelar acciones que intenta ejecutar. 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 cancelar, el **HPage Server** reinicia al **Hint Server**, lo configura según los demás parámetros (ej: `ghcOptions`) y vuelve a ejecutar el `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) [21]. Para ello utilizamos QuickCheck [29], una herramienta 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
```

3.3.4. UI

La interfaz gráfica de ***λPage*** está desarrollada utilizando *wxHaskell*, un framework elegido por ser multi-plataforma y, gracias a estar construído 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 [31].

4. Resultados

4.1. Objetivos Alcanzados

> Results! Why, man? I have gotten a lot of results. I know several thousand things that wont work
>
> Thomas A. Edison
>
> Kids, you tried your best and you failed miserably. The lesson is: "never try"
>
> Homer Simpsons

A primera vista, **λPage** puede parecer simplemente un editor 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, **λPage** 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"

Son todas estas características las que convierten a **λPage** 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.

4.2. 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

λPage 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 [6]. 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:

ByteStrings Las aplicaciones *Haskell* tradicionales manejan el texto utilizando **Strings** (o sea, listas de caracteres), pero ya se encuentra a disposición del desarrollador *Haskell* una nueva herramienta que permite manejar texto de una manera más óptima: **ByteStrings**. **λPage** utiliza en su mayoría **Strings** y podría ser optimizado migrando la mayor cantidad posible a **ByteStrings**.

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

Cabal La actual integración con *Cabal* cubre lo básico de la configuración de paquetes, como las extensiones y los módulos, pero *Cabal* permite describir varias cosas más que podrían ser aprovechadas por **λPage** como la ubicación de librerías y otros archivos

Auto Reload Una característica que sería muy útil agregar a **λPage** 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 **λPage** tome los cambios realizados.

5. 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. Entre ellas debemos agradecer especialmente a quienes 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 λ***Page*** con su herramienta *Hayoo!*. Y no podríamos dejar de mencionar a nuestros "beta-testers": Abram Hindle, Mariano Perez Rodriguez, Facundo Villanueva, Federico Grassi y Bernabé Panarello. Por último, pero no por eso menos importante, queremos agradecer a Darío Ruellan quien ha creado nuestra página web.

Personalmente yo, Fernando Benavides, quisiera agradecer a mi mujer, Constanza Zappala, que me ha acompañado, ayudado y soportado durante el más de un año y medio 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 *Algorítmos I* (Incluída 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 Garverbetsky, 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 con ellos.

Referencias

[1] Couchdb [online]. Available from: http://couchdb.apache.org.

[2] ejabberd [online]. Available from: http://www.ejabberd.im.

[3] Erlang [online]. Available from: http://www.erlang.org.

[4] Ghci [online]. Available from: http://www.haskell.org/ghc/docs/latest/html/users_guide/ghci.html.

[5] Ghci debugger [online]. Available from: http://www.haskell.org/ghc/docs/6.10-latest/html/users_guide/ghci-debugger.html.

[6] Github [online]. Available from: http://github.com.

[7] The glasgow haskell compiler [online]. Available from: http://www.haskell.org/ghc.

[8] Hackagedb [online]. Available from: http://hackage.haskell.org.

[9] Haskell [online]. Available from: http://www.haskell.org.

[10] The haskell cabal [online]. Available from: http://www.haskell.org/cabal.

[11] Haskell platform [online]. Available from: http://hackage.haskell.org/platform/.

[12] Hat - the haskell tracer [online]. Available from: http://www.haskell.org/hat.

[13] Hayoo! [online]. Available from: http://holumbus.fh-wedel.de/hayoo.

[14] Java scrapbook pages [online]. Available from: http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.jdt.doc.user/reference/ref-34.htm.

[15] Microsoft visual c++ 2008 sp1 redistributable package (x86) [online]. Available from: http://www.microsoft.com/downloads/details.aspx?familyid=A5C84275-3B97-4AB7-A40D-3802B2AF5FC2.

[16] Smalltalk [online]. Available from: http://www.smalltalk.org.

[17] Smalltalk workspace [online]. Available from: http://wiki.squeak.org/squeak/1934.

[18] Wxhaskell [online]. Available from: http://haskell.org/haskellwiki/WxHaskell.

[19] Wxwidgets [online]. Available from: http://www.wxwidgets.org/.

[20] Java [online]. October 2007. Available from: http://www.java.com.

[21] Kent Beck. Test Driven Development: By Example. Addison-Wesley Professional, 2002.

[22] Fernando Benavides. eprocess. http://hackage.haskell.org/package/eprocess, October 2009. Available from: http://hackage.haskell.org/package/eprocess.

[23] Fernando Benavides. λpage [online]. 2009. Available from: http://haskell.hpage.com.

[24] Fernando Benavides. wxhnotepad, December 2009. Available from: http://github.com/elbrujohalcon/wxhnotepad.

[25] Fernando Benavides. hfiar [online]. January 2010. Available from: http://hackage.haskell.org/package/hfiar.

[26] 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.

[27] Daniel Gorín. Hint. http://projects.haskell.org/hint, January 2009.

[28] Keith Hanna. A document-centered environment for haskell. April 2005. Available from: http://www.cs.kent.ac.uk/projects/vital/description/2005/all.ps.

[29] John Hughes Koen Claessen. Quickcheck [online]. Available from: http://www.cs.chalmers.se/~rjmh/QuickCheck/.

[30] Wilf R. LaLonde and John R. Pugh. Inside Smalltalk: vol. 1, volume I, page 71. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[31] Jeremy O'Donoghue. Building a text editor. http://wewantarock.wordpress.com, January 2010.

[32] Bryan O'Sullivan, John Goerzen, and Don Stewart. Real World Haskell. O'Reilly Media, Inc., 1 edition, 2008.

[33] Chris Piro. Facebook chat, February 2009. Available from: http://www.facebook.com/note.php?note_id=51412338919.