Tesis de Licenciatura

λ**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

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-dos, 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[3], todos ellos desarrollados utilizando lenguajes del paradigma funcional.

Ejemplos de éstos lenguajes de programación, como Haskell[4] o Erlang[5], 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"*[6] .

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

GHCi[7] La consola que provee GHC[8] permite a los desarrolladores evaluar expresiones, verificar su tipo o su clase. Cuenta también con un mecanismo de debugging[9] 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

[1] http://couchdb.apache.org
[2] http://www.ejabberd.im
[3] http://www.facebook.com
[4] http://www.haskell.org
[5] http://www.erlang.org
[6] Entiéndase "micro-testing" como la tarea de realizar tests eventuales para entender o evaluar algún aspecto de un programa
[8] http://www.haskell.org/ghc
[9] http://www.haskell.org/ghc/docs/6.10-latest/html/users_guide/ghci-debugger.html

Hat[10] 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[11] y características deseadas[12].

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 de este tipo, como Java Scrapbook Pages[13] para Java[14] y Workspace[15] 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[17] de la Universidad de Kent, ha desarrollado Vital[18]. *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 ***λPage*** hemos tenido que enfrentar varios desafíos relacionados principalmente con el desarrollo de interfaces visuales dentro del paradigma funcional. Volcando el conocimiento adquirido durante ese

[11] http://www.haskell.org/hat/bugs.html
[12] http://www.haskell.org/hat/bugs.html
[13] http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.jdt.doc.user/reference/ref-34.htm
[14] http://www.java.com
[15] http://wiki.squeak.org/squeak/1934
[16] http://www.smalltalk.org
[17] http://www.cs.kent.ac.uk/people/staff/fkh/
[18] http://www.cs.kent.ac.uk/projects/vital/

proceso, hemos desarrollado wxhNotepad[19] que es, ante todo, una prueba de concepto sobre cómo desarrollar editores de texto con *wxHaskell*. Gracias a Jeremy O'Donoghue[20], *wxhNotepad* está siendo publicado como un tutorial[21] en sucesivos artículos en su blog

1.3. *λPage*

> 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

λPage[22] 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. ***λPage*** 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*, ***λPage*** se integra con *Cabal*[23] y Hayoo![24] y se encuentra ya disponible en HackageDB[25].

λPage presenta una interfaz simple e intuitiva, desarrollada utilizando wxHaskell[26], lo que lo convierte en un sistema multiplataforma.

Por ser una herramienta desarrollada con *Haskell* para *Haskell*, ***λPage*** 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).

[19] http://github.com/elbrujohalcon/wxhnotepad
[20] http://wewantarock.wordpress.com/about/
[21] http://wewantarock.wordpress.com/2010/01/31/building-a-text-editor-part-1/
[22] http://haskell.hpage.com
[23] http://www.haskell.org/cabal
[24] http://holumbus.fh-wedel.de/hayoo
[25] http://hackage.haskell.org/package/hpage
[26] http://haskell.org/haskellwiki/WxHaskell

A diferencia de *GHCi* que es una herramienta "de consola", **λPage** 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.

λPage se encuentra desarrollado utilizando *eprocess*[27], una librería que facilita el manejo de "threads" en un estilo similar al de los procesos *Erlang*. Gracias al uso de esta librería, **λPage** 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*, **λPage** 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.

[27] http://hackage.haskell.org/package/eprocess

2. Tutorial - 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+[28] 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[29] 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[30].

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
```

[30] http://www.microsoft.com/downloads/details.aspx?familyid=A5C84275-3B97-4AB7-A40D-3802B2AF5FC2

2.2. QuickStart

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

> 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

TODO: Tutorial donde se noten las features de λ***Page***

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[31]

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 1. 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.

[31] http://projects.haskell.org/hint

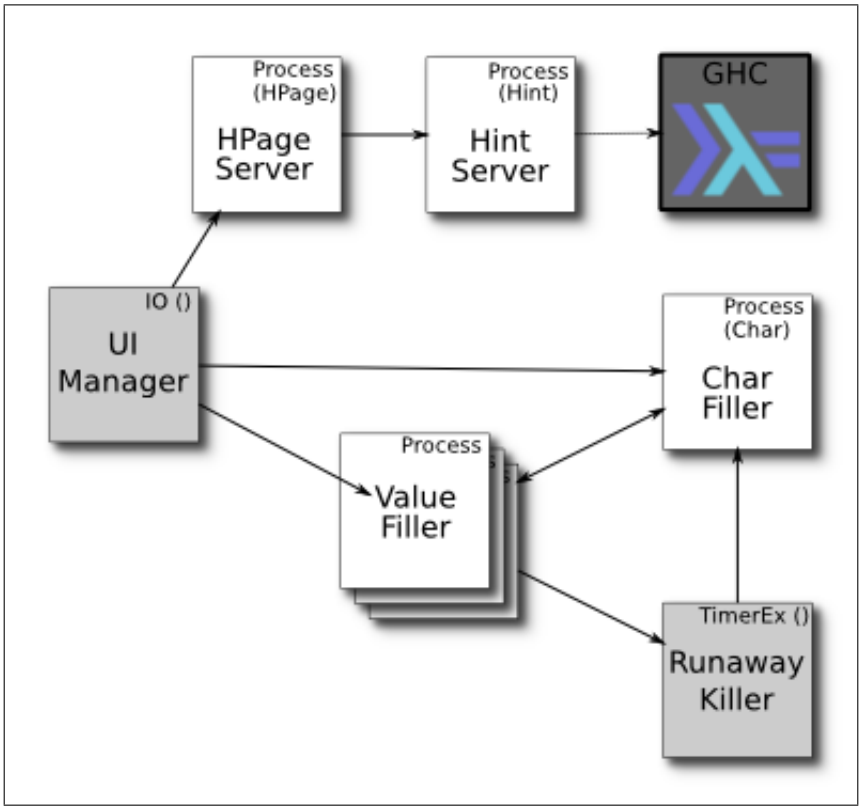

Figura 1: Arquitectura de λ***Page***

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 2 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

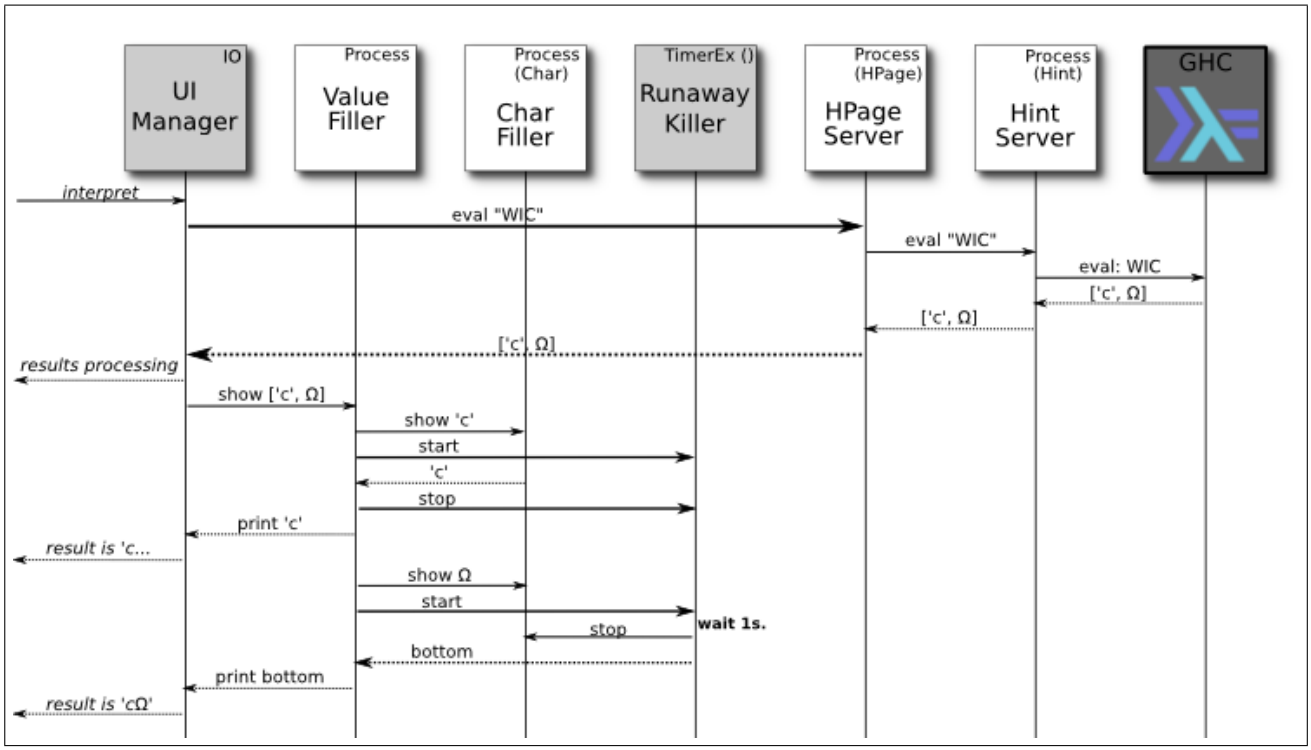

Figura 2: Secuencia de Evaluación de “WIC”

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*. ***λPage*** intenta entonces ejecutar la función `show` sobre 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 el 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
```

[32] http://book.realworldhaskell.org/read/concurrent-and-multicore-programming.html

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 *proceso* sino simplemente en cualquier instancia de la clase `MonadIO`.

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 λ***Page*** (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

```
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.

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 la lectura de los artículos escritos por Jeremy O'Donoghue en su tutorial Building a text editor[33].

TODO: Ver el código... detectar cosas interesantes y armarles secciones

[33] http://wewantarock.wordpress.com/2010/01/31/building-a-text-editor-part-1/

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

TODO: ¿Qué se puede hacer ahora que existe λ***Page***?

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

So much to do, so little done, such things to be TODO: Future Work

5. Agradecimientos

TODO: Acknowledgments. Recordar a Jeremy, al flaco del Undo/Redo, a los de Hayoo!