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

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

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] 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
[11] http://www.haskell.org/hat/bugs.html
[12] http://www.haskell.org/hat/bugs.html

1.2. Trabajos Relacionados

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").

1.3. λ*Page*

λ***Page***[19] 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[20] y Hayoo![21] y se encuentra ya disponible en HackageDB[22].

λ***Page*** presenta una interfaz simple e intuitiva, desarrollada utilizando wxHaskell[23], lo que lo convierte en un sistema 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

[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/
[19] http://haskell.hpage.com
[20] http://www.haskell.org/cabal
[21] http://holumbus.fh-wedel.de/hayoo
[22] http://hackage.haskell.org/package/hpage
[23] http://haskell.org/haskellwiki/WxHaskell

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", **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*[24], 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.

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

2. Tutorial - Descubriendo *λPage*

2.1. Instalación

Para instalar ***λPage*** en *OSX* o *Windows*, se proveen instaladores en el sitio web de ***λPage***, sin embargo,como se ha dicho, ***λPage*** 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+[25] 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[26] Una distribución de *Haskell* que incluye todo lo necesario para compilar e instalar programas desarrollados en este lenguaje (de particular interés para ***λPage***: GHC y happy).

2.1.1. Windows

Para el correcto funcionamiento de ***λPage*** los usuarios de *Windows XP* deben instalar el C++ 2008 SP1[27].

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
```

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

2.2. QuickStart

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

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

3.1. Arquitectura General

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[28]

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

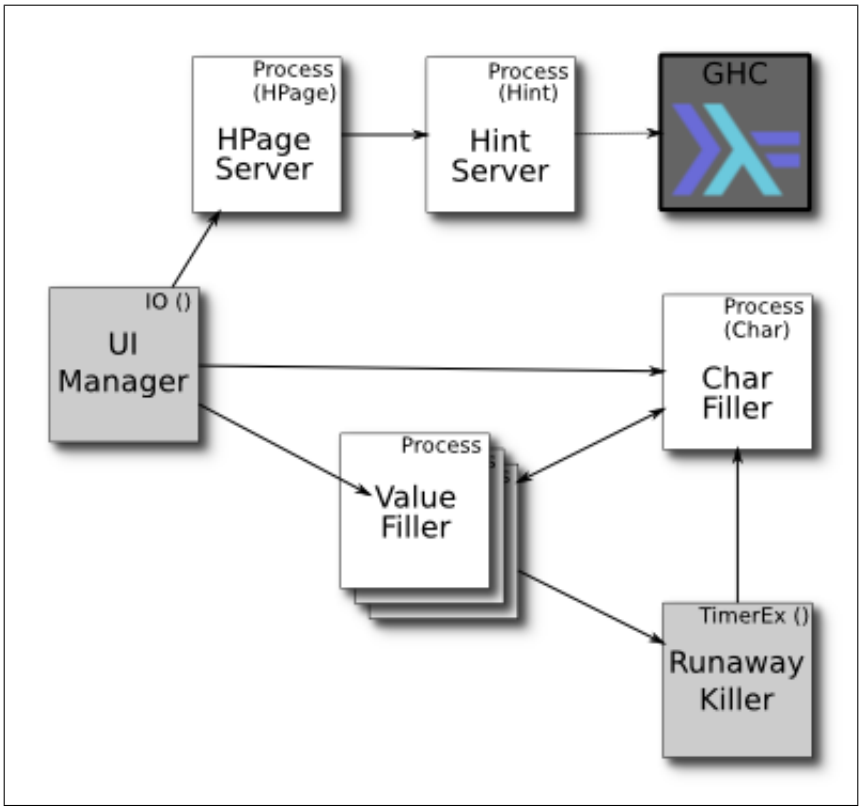

Figura 1: Arquitectura de ***λPage***

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

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.

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.

3.2. Diseño

TODO: Contar las decisiones que tomamos y por qué

3.3. Implementación

TODO: Detalles generales de implementación

3.3.1. wxHaskell

TODO: Pros y contras y workarounds

3.3.2. Bottoms

TODO: Cómo manejamos los bottoms en el resultado?

3.3.3. Threads

TODO: Cómo manejamos los threads para la GUI y la VM?

3.3.4. hint

TODO: Cómo utilizamos hint para conectarnos con la VM y el tema de que es *lazy*

3.4. Problemas Resueltos

3.4.1. Un Editor de Texto en Haskell

TODO: wxhNotepad

3.4.2. Multithreading en GHC

TODO: ¿Cómo simular multhreading cuando GHC no es multithread? TODO: Otros

4. Resultados

4.1. Objetivos Alcanzados

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

4.2. Trabajo a Realizar

TODO: Future Work