

λ Page

Fernando Benavides

Departamento de Computación, FCEyN, Universidad de Buenos Aires.

13 de julio de 2010

El Orador

¿quién soy?

- ▶ Fernando Benavides

¿cómo llegué hasta aquí?

- ▶ Alumno de Computación desde 2001
- ▶ Programador desde hace más de 10 años
- ▶ Programador *Funcional* desde hace 2 años

El Orador

¿quién soy?

- ▶ Fernando Benavides

¿cómo llegué hasta aquí?

- ▶ Alumno de Computación desde 2001
- ▶ Programador desde hace más de 10 años
- ▶ Programador *Funcional* desde hace 2 años

λ Page

Un bloc de notas para usuarios Haskell

Una herramienta para...

- ▶ debuggear
- ▶ entender código
- ▶ realizar *micro-testing*

λPage

Un bloc de notas para usuarios Haskell

Una herramienta para...

- ▶ debuggear
- ▶ entender código
- ▶ realizar *micro-testing*

Contexto

Presenciamos actualmente la aparición de varias aplicaciones desarrolladas dentro del *paradigma funcional*:

- ▶ CouchDB
- ▶ ejabberd
- ▶ Chat de Facebook

Desarrolladas en lenguajes como:

- ▶ Erlang
- ▶ Haskell

Contexto

Presenciamos actualmente la aparición de varias aplicaciones desarrolladas dentro del *paradigma funcional*:

- ▶ CouchDB
- ▶ ejabberd
- ▶ Chat de Facebook

Desarrolladas en lenguajes como:

- ▶ Erlang
- ▶ Haskell

Necesidades

Para desarrollar en *Haskell* existen herramientas como:

- ▶ GHCi
- ▶ Hugs
- ▶ Hat

Pero no existen herramientas como

- ▶ **Java** Scrapbook Pages
- ▶ Workspace de **Smalltalk**

Necesidades

Para desarrollar en *Haskell* existen herramientas como:

- ▶ GHCi
- ▶ Hugs
- ▶ Hat

Pero no existen herramientas como

- ▶ **Java** Scrapbook Pages
- ▶ Workspace de **Smalltalk**

λ Page

λ Page es una herramienta similar al Workspace de Smalltalk, en tanto:

- ▶ permite al desarrollador trabajar con texto libre
- ▶ detecta expresiones y definiciones válidas
- ▶ permite inspeccionarlas y evaluarlas

Además, λ Page, brinda otras facilidades particulares para *Haskell*:

- ▶ Integración con *Cabal* y *Hayoo*!
- ▶ Aprovecha *lazy evaluation* y *tipado estático*
- ▶ Presenta resultados dinámicamente

λPage

λPage es una herramienta similar al Workspace de Smalltalk, en tanto:

- ▶ permite al desarrollador trabajar con texto libre
- ▶ detecta expresiones y definiciones válidas
- ▶ permite inspeccionarlas y evaluarlas

Además, *λPage*, brinda otras facilidades particulares para *Haskell*:

- ▶ Integración con *Cabal* y *Hayoo*!
- ▶ Aprovecha *lazy evaluation* y *tipado estático*
- ▶ Presenta resultados dinámicamente

Interpretación de Expresiones

λPage permite interpretar expresiones como:

```
[1 , 2 , 3]
```

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

```
[1 , 2 ..]
```

```
let loop = loop in 1:loop
```

Interpretación de Expresiones

λPage permite interpretar expresiones como:

```
[1 , 2 , 3]
```

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

```
[1 , 2 .. ]
```

```
let loop = loop in 1:loop
```

Interpretación de Expresiones

λPage permite interpretar expresiones como:

```
[1 , 2 , 3]
```

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

```
[1 , 2 ..]
```

```
let loop = loop in 1:loop
```

Interpretación de Expresiones

λPage permite interpretar expresiones como:

```
[1 , 2 , 3]
```

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

```
[1 , 2 ..]
```

```
let loop = loop in 1:loop
```

Presentación de Resultados

- ▶ Para expresiones inválidas, λ Page presenta el error informado por *GHC*
- ▶ Para expresiones sin resultado “visible”, λ Page permite conocer su tipo
- ▶ Para expresiones infinitas, λ Page presenta su valor incrementalmente hasta que el usuario cancela la evaluación
- ▶ Para expresiones que requieren cálculos infinitos, λ Page permite al usuario cancelar la evaluación

Acciones con Efectos Colaterales

Para realizar acciones que puedan generar efectos colaterales, en *Haskell* se utiliza la mónada *IO*. Por ejemplo:

```
readFile 'README' :: IO String
```

IO String no es un tipo “visible”, por lo que el valor de la expresión no se podría mostrar.

Pero *λPage* toma el modelo de *GHCi* y ejecuta la acción, presentando su resultado

Acciones con Efectos Colaterales

Para realizar acciones que puedan generar efectos colaterales, en *Haskell* se utiliza la mónada *IO*. Por ejemplo:

```
readFile 'README' :: IO String
```

IO String no es un tipo “visible”, por lo que el valor de la expresión no se podría mostrar.

Pero *λPage* toma el modelo de *GHCi* y ejecuta la acción, presentando su resultado

Acciones con Efectos Colaterales

Para realizar acciones que puedan generar efectos colaterales, en *Haskell* se utiliza la mónada *IO*. Por ejemplo:

```
readFile 'README' :: IO String
```

IO String no es un tipo “visible”, por lo que el valor de la expresión no se podría mostrar.

Pero *λPage* toma el modelo de *GHCi* y ejecuta la acción, presentando su resultado

Listas

λPage interpreta de modo particular las listas (expresiones de tipo `Show a => [a]`)

Tomemos como ejemplo la siguiente expresión:

```
let loop = loop in [1, div 0 0, 2,  
                     undefined, 3, loop, 4]
```

Listas

```
let loop = loop in [1, div 0 0, 2,  
                    undefined, 3, loop, 4]
```

Si *λPage* presentase su evaluación tal como lo hace con las demás expresiones, el resultado sería

```
[1,
```

e informaría al usuario la excepción encontrada (o sea, `DivideByZero`)

Listas

```
let loop = loop in [1, div 0 0, 2,  
                    undefined, 3, loop, 4]
```

λPage en cambio, podría evaluar cada elemento por separado y detectar excepciones. En tal caso, el resultado sería

```
[1, ⊥, 2, ⊥, 3,
```

y continuaría intentando calcular el siguiente elemento hasta que el usuario decidiese cancelar

Listas

```
let loop = loop in [1, div 0 0, 2,  
                    undefined , 3, loop , 4]
```

λPage, sin embargo, detecta cálculos posiblemente infinitos y presenta como resultado:

```
[1, ⊥, 2, ⊥, 3, ⊥, 4]
```

permitiendo luego al usuario conocer el motivo de cada \perp a través de un menú contextual

Paralelismo

Muchas cosas suceden al mismo tiempo en *λPage*

- ▶ Manejo de Páginas (crear, abrir, modificar, guardar, cerrar, etc.)
- ▶ Interpretación de Expresiones
- ▶ Evaluación de Acciones de Entrada/Salida

Para lograrlo, creamos *eprocess*:

- ▶ Basada conceptualmente en *Erlang*
- ▶ Construida utilizando *Threads*, *Channels* y *MVars*

Paralelismo

Muchas cosas suceden al mismo tiempo en *λPage*

- ▶ Manejo de Páginas (crear, abrir, modificar, guardar, cerrar, etc.)
- ▶ Interpretación de Expresiones
- ▶ Evaluación de Acciones de Entrada/Salida

Para lograrlo, creamos *eprocess*:

- ▶ Basada conceptualmente en *Erlang*
- ▶ Construída utilizando *Threads*, *Channels* y *MVars*

Manejo de Módulos

Utilizando *λPage*, el usuario puede

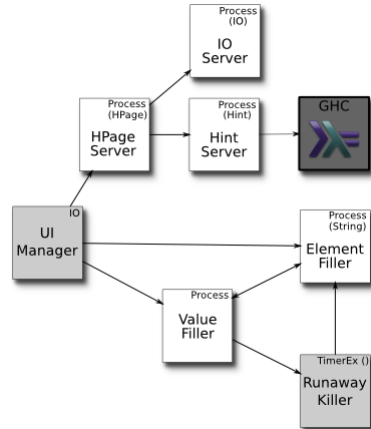
- ▶ Importar módulos
- ▶ Cargar módulos
- ▶ Recargar módulos
- ▶ Ver los módulos de un paquete *Cabal*
- ▶ Ver la documentación de un módulo utilizando *Hayoo!*

Todo esto *sin perder las expresiones que ya tiene definidas*

Arquitectura

Principales Requerimientos:

- Conexión con GHC
- Paralelismo
- Errores Controlados
- Presentación de Resultados



Componentes

UI Manager Administrador de la GUI

HPage Server Interfaz entre la GUI y *GHC*

IO Server Para ejecutar acciones de entrada/salida

Hint Server Conecta *HPage* y *GHC*

Value Filler Encargado de procesar y mostrar expresiones

Element Filler Encargado de evaluar elementos

Runaway Killer Detector de cálculos infinitos

Principales Decisiones de Diseño

- ▶ Creación de *eprocess*
- ▶ Creación de **servers**
- ▶ Presentación de Interpretaciones
 - ▶ Entrada/Salida
 - ▶ Listas
 - ▶ Tipos
 - ▶ Expresiones “visibles”
 - ▶ Expresiones “no visibles”
- ▶ Integración con *Cabal* y *Hayoo*!

eprocess

```
newtype ReceiverT r m a
```

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

```
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
```

eprocess

```
newtype ReceiverT r m a
```

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

```
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
```

eprocess

```
newtype ReceiverT r m a
```

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

```
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
```


Servers (IOServer)

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

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

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

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

HPage.Control

```
newtype Expression = Exp {exprText :: String}  
    deriving (Eq, Show)  
  
data Page = Page { — Display —  
    expressions :: [Expression],  
    currentExpr :: Int,  
    undoActions :: [HPage ()],  
    redoActions :: [HPage ()],  
    — File System —  
    original :: [Expression],  
    filePath  :: Maybe FilePath  
}
```

HPage.Control

```
data Context = Context { — Package —  
  activePackage :: Maybe PackageIdentifier ,  
  pkgModules   :: [Hint.ModuleName] ,  
  — Pages —  
  pages        :: [Page] ,  
  currentPage  :: Int ,  
  — GHC State —  
  loadedModules :: Set String ,  
  importedModules :: Set String ,  
  extraSrcDirs  :: [FilePath] ,  
  ghcOptions    :: String ,  
  server        :: HS.ServerHandle ,  
  ioServer      :: HPIO.ServerHandle ,  
  — Actions —  
  recoveryLog   :: Hint.InterpreterT IO () }
```

Objetivos Alcanzados

λPage permite al usuario

- ▶ Configurar de entorno según paquetes *Cabal*
- ▶ Editar páginas mientras se evalúa una expresión
- ▶ Visualizar expresiones con errores o cálculos infinitos sin bloquearse mostrando el resultado más completo posible
- ▶ Importar, cargar y recargar módulos sin perder las expresiones con las que está trabajando
- ▶ Intercalar definiciones, expresiones y texto libre

Logros Adicionales

Más allá de los objetivos propuestos para esta tesis, *λPage* permite

- ▶ Consultar la API de *Haskell* utilizando *Hayoo!*
- ▶ Trabajar en *OSX*, *Windows* y *Linux*
- ▶ Administrar varias páginas con expresiones, pudiendo cargarlas, guardarlas, etc.
- ▶ Determinar el contexto de evaluación dinámicamente
- ▶ Navegar módulos importados o cargados

Tareas a Realizar

- ▶ Mejoras visuales
- ▶ Presentación de resultados:
 - ▶ Nuevas visualizaciones
 - ▶ Distintos tipos a tratar de modo particular
 - ▶ Composición
- ▶ Otras herramientas
 - ▶ Soporte para TDD
 - ▶ Refactoring
 - ▶ Análisis de Terminación
 - ▶ Debugging
- ▶ Otros lenguajes
 - ▶ Erlang

Tareas a Realizar

- ▶ Mejoras visuales
- ▶ Presentación de resultados:
 - ▶ Nuevas visualizaciones
 - ▶ Distintos tipos a tratar de modo particular
 - ▶ Composición
- ▶ Otras herramientas
 - ▶ Soporte para TDD
 - ▶ Refactoring
 - ▶ Análisis de Terminación
 - ▶ Debugging
- ▶ Otros lenguajes
 - ▶ Erlang

Tareas a Realizar

- ▶ Mejoras visuales
- ▶ Presentación de resultados:
 - ▶ Nuevas visualizaciones
 - ▶ Distintos tipos a tratar de modo particular
 - ▶ Composición
- ▶ Otras herramientas
 - ▶ Soporte para TDD
 - ▶ Refactoring
 - ▶ Análisis de Terminación
 - ▶ Debugging
- ▶ Otros lenguajes
 - ▶ Erlang

Tareas a Realizar

- ▶ Mejoras visuales
- ▶ Presentación de resultados:
 - ▶ Nuevas visualizaciones
 - ▶ Distintos tipos a tratar de modo particular
 - ▶ Composición
- ▶ Otras herramientas
 - ▶ Soporte para TDD
 - ▶ Refactoring
 - ▶ Análisis de Terminación
 - ▶ Debugging
- ▶ Otros lenguajes
 - ▶ Erlang

¡Gracias a todos!

¿En qué los puedo ayudar?