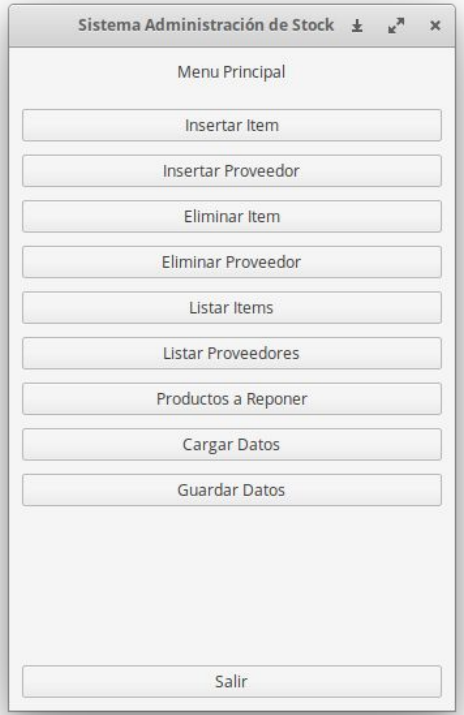
GTK+ Declarativo con Haskell

Emanuel Lautaro - 2020

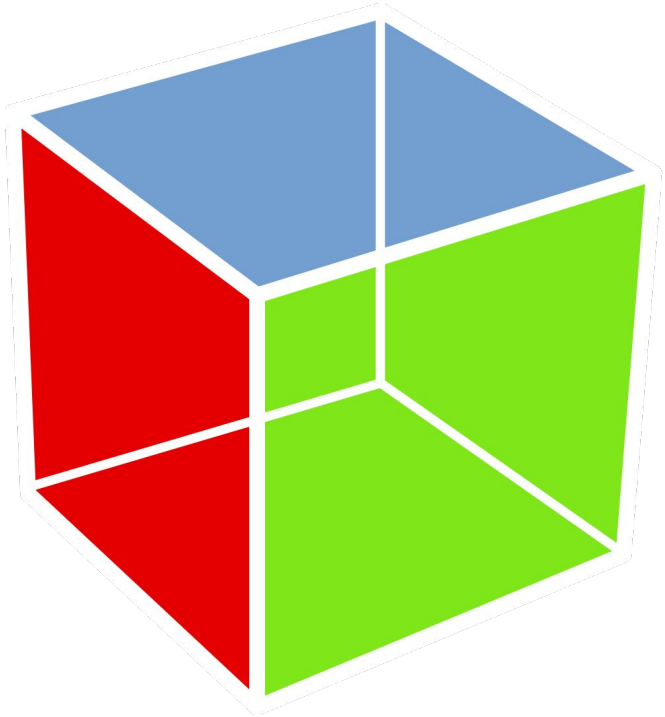
Proyecto: Administración de Stock

- Administrador CRUD de Ítems y Proveedores
- Escrito totalmente en Haskell
- Aplicación de Escritorio (GUI)
- Objetivos:
  - Vista y Modelo desacoplados
  - Funcional
  - Pureza (sin mutaciones ni efectos secundarios)
  - Visualmente agradable



GTK+

- GTK+ idiomático
  - Crear widgets de manera imperativa
  - “Setear” atributos
  - Añadir handlers (callbacks) a eventos
- GTK+ Glade
  - Lenguaje de enmarcado XML
  - Editor visual
  - Prototipado
  - Primera vista declarativa con cambios imperativos
- Soporte limitado para CSS



Problemas del Enfoque Imperativo

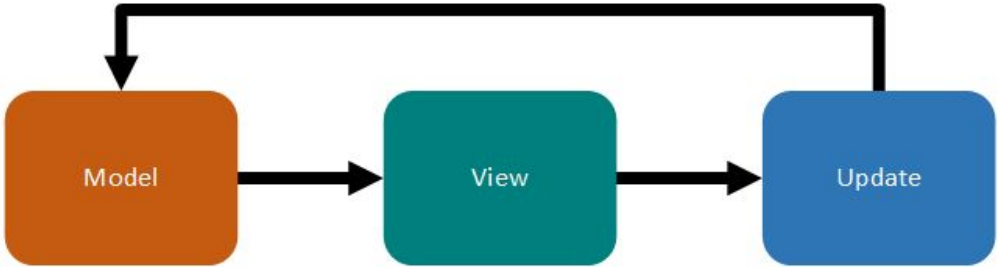
- Cuando la aplicación crece, los “callbacks” se vuelven inmanejables
  - Uso de `IORefs` (variables mutables), lógica y estado por doquier
  - Efectos secundarios
  - Dificultad para el testing
- El código de la `Vista` debería ser
  - Independiente de la lógica de negocios
  - Declarativa
  - Concisa
- La lógica de negocio debería ser
  - Pura (idealmente)
  - Desacoplada de la `Vista`

Programación Declarativa

Arquitectura MVU

Utilizada en frameworks populares y modernos

- **React + Redux** (Web, Android y iOS)
- **Flutter** (Android y iOS)
- **Jetpack Compose** (Android)

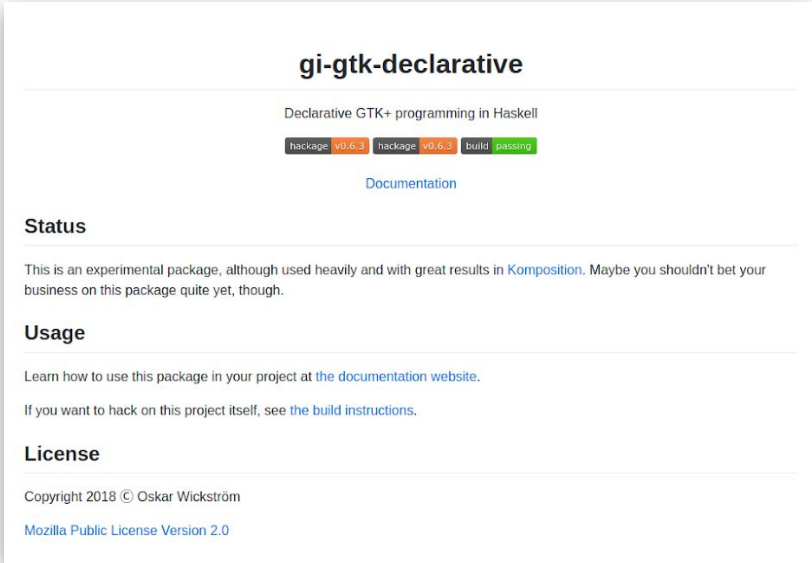


GTK+ Declarativo

- Vistas declarativas, usando funciones de `Modelo -> Vista`
- Manejo de eventos declarativos sin concurrencia
- Actualizaciones a la `Vista` usando un “DOM Virtual”
- Totalmente desacoplado de la arquitectura
- Inspirado en la **Arquitectura Elm** y **Purescript Pux (MVU)**

gi-gtk-declarative

- github.com/owickstrom/gi-gtk-declarative
- Capa de abstracción sobre **gi-gtk**
- Funcionalidades avanzadas de **GHC**: `OverloadedLabels`, Tipos Avanzados ...
- Soporte para gran cantidad de Widgets de GTK+



Widgets Simples

- Los `Widgets` simples (sin hijos) se construyen utilizando `widget`:

```
widget Button []

widget CheckButton []

widget Label []

...
```

Bins

- En GTK+, un `bin` puede contener **exactamente** un solo `widget` hijo
- Algunos `bins` son

    `ScrolledWindow`

    `Viewport`

    `SearchBar`

- Para asignar múltiples hijos a un `bin` se debe utilizar un `container` intermedio

Containers

- Un `container` pueden contener cero o más widgets hijos
- Algunos `containers` restringen los tipos de hijos
  - `ListBox` requiere que los hijos sean de tipo `ListBoxRow`:

```
container ListBox [] [
  bin ListBoxRow [] $ widget Button [],
  bin ListBoxRow [] $ widget CheckButton []]
```

  - Box requiere que los hijos sean de tipo `BoxChild`:

```
container Box [] [
  BoxChild {..} $ widget Button [],
  BoxChild {..} $ widget CheckButton []]
```

Atributos

- Atributos de `Widgets:`

```
widget Button [#label := "Pulse Aquí"]
```

- Atributos de `Bins:`

```
bin ScrolledWindow [#hscrollbarPolicy := PolicyTypeAutomatic ] $ listView
```

- Atributos de `Containers:`

```
container ListBox [#selectionMode := SelectionModeMultiple ] $ children
```

Eventos

- La función on permite emitir eventos:

```
botonContador cantidadClicks =
let msg = "Me han clickeado "
          <> cantidadClicks
          <> " veces."
 in widget
     Button
       [#label := msg
       ,on #clicked ClickBoton]
```

Eventos con IO

- Algunos eventos requieren de IO para obtener datos de los widgets

```
botonColor color =
  widget
     ColorButton
       [#title := "Color seleccionado"
       ,#rgba := color
       ,onM #colorSet (fmap CambioColor . getColorButtonRgba)]
```

Functores

- `Widget`, `Bin`, y `Container` son instancias de `Functor`, pudiendo anidar eventos

```
data EventoBoton = BotonPresionado

miBoton :: Text -> Widget EventoBoton

data MiEvento = Incr | Decr

botonIncrDecr :: Widget MiEvento
botonIncrDecr =
  container Box [#orientation := OrientationHorizontal]
    [BoxChild {..} $ miBoton "-1" $> Decr
    ,BoxChild {..} $ miBoton "+1" $> Incr]
```

Estilos con CSS

- Se pueden añadir clases **CSS** utilizando classes:

```
widget Button [classes ["botón-primario"], #label := "Aceptar"]

widget Label [classes ["titulo"], #label := "Mi Aplicación"]
```

“Hola Mundo”

“Hola Mundo” imperativo

```
main :: IO ()
main = do
  _ <- Gtk.init Nothing
  win <- new Gtk.Window [#title := "¡Hola!"]
  #resize win 200 150
  _ <- on win #destroy Gtk.mainQuit
  button <- new Gtk.Button [#label := "Presiona Aquí"]
  _ <- on button
      #clicked
      (set button [#sensitive := False, #label := "¡Sorpresa!"])
  #add win button
  #showAll win
  Gtk.main
```

El Modelo y los Eventos

```
data Modelo = SinPresionar | Presionado

data Evento = BotonPresionado
```

La Vista

```
view' :: Model -> Widget Evento
view' model =
  bin Window
     [#title := "¡Hola!", #widthRequest := 200, #heightRequest := 150] $
  case model of
     SinPresionar ->
       widget Button [#label := "Presiona Aquí", on #clicked BotonPresionado]
     Presionado ->
       widget Button [#sensitive := False, #label := "¡Sorpresa!"]
```

Función ‘update’

```
update' :: Modelo -> Evento -> Transition Modelo Evento
update' _ BotonPresionado = Transition Presionado (pure Nothing)
```

El último parámetro es de tipo

`IO (Maybe Evento)`

Podemos ejecutar código en `IO` en cada llamada a `update’`

Código ‘main’

```
main :: IO ()
main = void $ run App
     { view = view'
     , update = update'
     , inputs = []
     , initialState = SinPresionar}
```

Pros y Contras

Pros

- Vista y lógica de negocios totalmente desacopladas
- Composición absoluta de widgets
- Acceso a **Haskell** y su ecosistema
- Codificación de reglas de negocio en los tipos
- Fácil razonamiento ante la ausencia de variables mutables y/o efectos secundarios
- Sobre hombros de gigantes: GTK+
- Aplicaciones nativas y multiplataforma

Contras

- Es necesario propagar el estado actual a todos los widgets de la `Vista`
- Se debe declarar todo el estado de la aplicación en el `Modelo`
- La función `update` crece rápidamente
- La arquitectura resulta extraña para programadores imperativos
- No hay gran cantidad de documentación ni ejemplos

En Resumen

- Programación basada en callbacks y/o observables es compleja
- Es mejor usar funciones puras y datos para modelar la lógica de negocio
- Obtener una función de `Modelo -> Widget` es **ideal**
- GTK+ puede ser usado de manera declarativa con **gi-gtk-declarative**
  - Librería altamente experimental
  - Ausencia de `Widgets` claves (ej. `TreeView`)
  - Se debe recurrir al modo imperativo para ciertos casos de uso (ej. `Dialog`)
- **GHC** es un aliado, no un enemigo
- Es posible usar **Haskell** para aplicaciones reales

¿Preguntas?

Todo el código se encuentra disponible en: github.com/emlautarom1/AdministracionStock