GUI Programming with Function Logic Languages

Michael Hanus

Christian-Albrechts-University of Kiel
Programming Languages and Compiler Construction

PSU 2008
Applications of FLP

General advantage of functional logic programming:
high-level interfaces for application libraries
Applications of FLP

General advantage of functional logic programming:
high-level interfaces for application libraries

- GUIs
- web programming
- databases
- distributed programming
- ...
Applications of FLP

General advantage of functional logic programming:
high-level interfaces for application libraries

- GUIs
- web programming
- databases
- distributed programming
- ...

Necessary: some knowledge about declarative I/O programming
Problem: Handling input/output in a declarative manner?
Problem: Handling input/output in a declarative manner?

Solution: External world as parameter to all I/O operations (Haskell, Mercury, Curry)
Declarative Input/Output

Problem: Handling input/output in a declarative manner?

Solution: External world as parameter to all I/O operations (Haskell, Mercury, Curry)

I/O action: transformation on the external world

Type of I/O actions (conceptually)

\[ \text{IO} \ a \ \approx \ \text{World} \rightarrow (a, \text{World}) \]

Interactive program: sequence(!) of actions applied to external world

Note: the “world” is implicit parameter, not explicitly accessible!
Some primitive I/O actions

getChar :: IO Char  -- read char from stdin
putChar :: Char -> IO () -- write argument to stdout
return :: a -> IO a -- do nothing, return arg.

getChar applied to a world  \rightarrow  character + new (transformed) world
Declarative Input/Output: Primitives

Some primitive I/O actions

\[
\begin{align*}
\text{getChar} &: \text{IO Char} \quad \text{-- read char from stdin} \\
\text{putChar} &: \text{Char} \rightarrow \text{IO } () \quad \text{-- write argument to stdout} \\
\text{return} &: \text{a} \rightarrow \text{IO a} \quad \text{-- do nothing, return arg.}
\end{align*}
\]

\text{getChar} \text{ applied to a world } \leadsto \text{ character + new (transformed) world}

Compose actions

\[
(\gg\gg\geq) \,: \text{IO a} \rightarrow (\text{a} \rightarrow \text{IO b}) \rightarrow \text{IO b}
\]

\text{getChar} \gg\gg\geq \text{putChar} \approx \text{ copy character from input to output}
### Some primitive I/O actions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getChar :: IO Char</code></td>
<td>-- read char from stdin</td>
</tr>
<tr>
<td><code>putChar :: Char -&gt; IO ()</code></td>
<td>-- write argument to stdout</td>
</tr>
<tr>
<td><code>return :: a -&gt; IO a</code></td>
<td>-- do nothing, return arg.</td>
</tr>
</tbody>
</table>

`getChar` applied to a world $\leadsto$ character + new (transformed) world

### Compose actions

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(&gt;&gt;=) :: IO a -&gt; (a -&gt; IO b) -&gt; IO b</code></td>
<td>copy character from input to output</td>
</tr>
</tbody>
</table>

### Specialized composition: ignore result of first action

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(&gt;&gt;) :: IO a -&gt; IO b -&gt; IO b</code></td>
<td>copy character from input to output</td>
</tr>
</tbody>
</table>

$x >> y = x >>= \_ \_ \_ \_ \_ -> y$
Declarative Input/Output: Examples

Output action for strings $\text{(String} \approx [\text{Char}])$

```haskell
putStr :: String -> IO ()
putStr [] = return ()
putStr (c:cs) = putChar c >> putStr cs
```

Advantage:
Standard programming techniques also applicable to I/O actions!

Apply an I/O action twice

```haskell
dupIO :: IO() -> IO ()
dupIO action = action >> action
```

dupIO (putStr "Ha ")
⇝ Output: Ha Ha

Imperative languages are not so funny!

Michael Hanus (CAU Kiel) GUI Programming with FL Languages PSU 2008
Declarative Input/Output: Examples

Output action for strings \((\text{String} \approx [\text{Char}])\)

\[
\begin{align*}
\text{putStr} :: & \text{String} \rightarrow \text{IO} () \\
\text{putStr} \; [] & = \text{return} () \\
\text{putStr} \; (c : cs) & = \text{putChar} \; c \; >> \; \text{putStr} \; cs
\end{align*}
\]

Advantage:
Standard programming techniques also applicable to I/O actions!
Declarative Input/Output: Examples

Output action for strings (String \approx [Char])

\textbf{putStr} :: \text{String} \rightarrow \text{IO ()}
\text{putStr} [] = \text{return ()}
\text{putStr} (c:cs) = \text{putChar} c \gg \text{putStr} cs

\textbf{Advantage:}
Standard programming techniques also applicable to I/O actions!

Apply an I/O action twice

\textbf{dupIO} :: \text{IO()} \rightarrow \text{IO ()}
dupIO \text{ action} = \text{action} \gg \text{action}
dupIO (\text{putStr} "Ha ") \leadsto \text{Output: Ha Ha}
Declarative Input/Output: Examples

Output action for strings \((\text{String} \approx [\text{Char}])\)

\[
\text{putStr :: String} \rightarrow \text{IO()} \\
\text{putStr} [] = \text{return} () \\
\text{putStr} (c:cs) = \text{putChar} c \gg \text{putStr} cs
\]

Advantage:
Standard programming techniques also applicable to I/O actions!

Apply an I/O action twice

\[
\text{dupIO :: IO()} \rightarrow \text{IO}() \\
\text{dupIO} \text{ action} = \text{action} \gg \text{action}
\]

\[
\text{dupIO (putStr "Ha ")} \leadsto \text{Output: Ha Ha}
\]

Imperative languages are not so funny!
Declarative Input/Output: Examples

Read a line

getLine :: IO String
getLine = getChar >>= \c ->
    if c=='\n' then return []
    else getLine >>= \cs -> return (c:cs)
Declarative Input/Output: Examples

Read a line

getLine :: IO String
getLine = getChar >>= \c ->
    if c=='\n' then return []
    else getLine >>= \cs -> return (c:cs)

Monadic composition not well readable  \rightarrow syntactic sugar
Haskell’s do notation

\[
\begin{align*}
\text{do } & p \leftarrow a_1 \quad \approx \quad a_1 >>= \ \langle p \rightarrow a_2 \\
a_2
\end{align*}
\]
Haskell’s do notation

\[
\begin{align*}
\text{do } & \quad p \leftarrow a_1 \quad \approx \quad a_1 \gg= \ \backslash p \quad \rightarrow \quad a_2 \\
 & \quad a_2
\end{align*}
\]

Again: Read a line

```
getLine = do c <- getChar
                if c==’\n’ then return []
                else do cs <- getLine
                        return (c:cs)
```
Haskell’s do notation

\[
\begin{align*}
do & \quad p \gets a_1 \\
& \quad a_2 \\
\end{align*}
\approx \quad a_1 \gg= \ \backslash p \rightarrow a_2
\]

Again: Read a line

```
getLine = do c <- getChar
           if c=='\n' then return []
           else do cs <- getLine
                   return (c:cs)
```

Note: no I/O in nondeterministic computations ("cannot copy the world")

\[\Rightarrow\] encapsulate search between I/O actions
Applications: GUI Programming

Graphical User Interfaces (GUIs)

- layout structure: hierarchical structure
- algebraic data type
- logical structure: dependencies in structure
- logic variables
- event handlers
- functions associated to layout structures

Advantages: compositional, less error prone

Michael Hanus (CAU Kiel) GUI Programming with FL Languages PSU 2008
Applications: GUI Programming

Graphical User Interfaces (GUIs)

- layout structure: hierarchical structure $\leadsto$ algebraic data type
- logical structure: dependencies in structure $\leadsto$ logic variables
- event handlers $\leadsto$ functions associated to layout structures
- advantages: compositional, less error prone
Applications: GUI Programming

Graphical User Interfaces (GUIs)

- layout structure: algebraic data type
- Data type for widgets: `Widget = Button ... Canvas CheckButton Entry Label ListBox Message ... Scale Int Int TextEdit Row Col`
Graphical User Interfaces (GUIs)

- layout structure: algebraic data type
Graphical User Interfaces (GUIs)

- layout structure: algebraic data type

### Data type for widgets

```haskell
data Widget = Button ...
| Canvas       [ConfItem]
| CheckButton  [ConfItem]
| Entry        [ConfItem]
| Label        [ConfItem]
| ListBox      [ConfItem]
| Message      [ConfItem]
| Scale Int Int [ConfItem]
| EditText     [ConfItem]
| Row          [...] [Widget]
| Col          [...] [Widget]
```
Applications: GUI Programming

Graphical User Interfaces (GUIs) layout structure: algebraic data type

Structure of counter GUI:

- Col [Entry [Text "0", Background "yellow"]
- Row [Button (...) [Text "Increment"], Button (...) [Text "Reset"], Button (...) [Text "Stop"]]

Missing: activities associated to button events
Graphical User Interfaces (GUIs)

- layout structure: algebraic data type
Applications: GUI Programming

Graphical User Interfaces (GUIs)

- layout structure: algebraic data type

Structure of counter GUI

| Col [] | [Entry [Text "0", Background "yellow"], |
| Row [] | [Button (...) [Text "Increment"], |
|        | Button (...) [Text "Reset"], |
|        | Button (...) [Text "Stop"] | ]

![Counter Demo GUI](image-url)
Applications: GUI Programming

Graphical User Interfaces (GUIs)

- layout structure: algebraic data type

Structure of counter GUI

Col [] [Entry [Text "0", Background "yellow"],
Row [] [Button (...) [Text "Increment"],
        Button (...) [Text "Reset"],
        Button (...) [Text "Stop"] ]]

Missing: activities associated to button events
Applications: GUI Programming

Graphical User Interfaces (GUIs)

- layout structure: hierarchical structure $\mapsto$ algebraic data type

Specification of a counter GUI

```
Col [Entry [  Text "0", Background "yellow"],
      Row [Button (    ) [Text "Increment"],
            Button (    ) [Text "Reset"],
            Button [Text "Stop"] ] ]
```
Graphical User Interfaces (GUIs)

- layout structure: hierarchical structure $\leadsto$ algebraic data type
- logical structure: dependencies in structure $\leadsto$ logic variables

Specification of a counter GUI

Col [Entry [WRef val, Text "0", Background "yellow"],
Row [Button ( val) [Text "Increment"],
   Button ( val) [Text "Reset"],
   Button [Text "Stop"] ]]

where val free
Graphical User Interfaces (GUIs)

- layout structure: hierarchical structure $\leadsto$ algebraic data type
- logical structure: dependencies in structure $\leadsto$ logic variables
- event handlers $\leadsto$ functions associated to layout structures

Specification of a counter GUI

Col [Entry [WRef val, Text "0", Background "yellow"],
        Row [Button (updateValue incr val) [Text "Increment"],
                Button (setValue val "0") [Text "Reset"],
                Button exitGUI [Text "Stop"] ]]

where val free
Specification of a counter GUI

Col [Entry [WRef \textit{val}, Text "0", Background "yellow"],
Row [Button (updateValue incr \textit{val}) [Text "Increment"],
Button (setValue \textit{val} "0") [Text "Reset"],
Button exitGUI [Text "Stop"] ]]

where \textit{val} free
Applications: GUI Programming

Specification of a counter GUI

Col [Entry [WRef val, Text "0", Background "yellow"],
     Row [Button (updateValue incr val) [Text "Increment"],
          Button (setValue val "0") [Text "Reset"],
          Button exitGUI [Text "Stop"] ]]
where val free

Some type signatures:

Button :: (GuiPort -> IO ()) -> [ConfItem] -> Widget
exitGUI :: GuiPort -> IO ()
setValue :: WidgetRef -> String -> GuiPort -> IO ()
getValue :: WidgetRef -> GuiPort -> IO String

GuiPort: (abstract) reference to GUI window
WidgetRef: (abstract) reference to widget in a GUI window
GUI Programming in FLP

Advantages:

- hierarchical specification of GUI layout
- dependencies for events as logic variables
- references checked by compiler (in contrast to Tcl/Tk)
- event handlers as functions associated to buttons

⇝ declarative specification of GUIs