Logic, Categories, and Graphical User Interfaces

Jennifer Paykin, Steve Zdancewic, Neel Krishnaswami

Wesleyan University
4/21/2015
GUIs

Clicked 0 times
GUIs

On click: run updateCounter code

Widgets

Callbacks

Clicked 0 times
A Simple GUI

n = 0
text = "Clicked " + str(n) + " times"

# button is a widget
button = Button(label = text,
               command=updateCounter)

# updateCounter : Unit -> Void
def updateCounter():
    n.set(n.get()+1)
    text.set("Clicked" + str(n) + "times")

mainloop()
Event Loop

- wait for event
- updateCounter: Data → ⊥
  - execute callback
  - pick a callback
  - onEvent: (Data → ⊥) → ⊥
  - execute callback
Non-local Code

Three sections of code:

1. Define widgets
2. Define callbacks
3. Define event loop
Non-local Code

Three sections of code:

1. Define widgets
2. Define callbacks
3. Define event loop

Surface language with one section
Event Loop

wait for event

pick a callback

updateCounter:Data → ⊥

execute callback

onEvent:(Data → ⊥) → ⊥
execute
callback
pick a
callback
onEvent:(Data
→
⊥)
→
⊥

Event Loop
wait for event
updateCounter:Data
→
⊥

execute callback
pick a callback
updateCounter:Data
→
⊥
onEvent:(Data
→
⊥)
→
⊥
Event Loop

wait for event

updateCounter: Data → ⊥

execute callback

onEvent: (Data → ⊥) → ⊥
Event Loop

wait for event

updateCounter: Data \rightarrow \bot

onEvent: (Data \rightarrow \bot) \rightarrow \bot
let counter =
  let n = 0 in
  let w = newWidget() in
  wait () = onClick(w) in
  wait () = drawButton(w, n+1) in
  ()
letrec count (w:Widget) (n:Nat) =
  wait () = onClick(w) in
  wait () = drawButton(w,n+1) in
  count w (n+1)

let counter () =
  let n = 0 in
  let w = newWidget() in
  count w n
letrec count (w:Widget) (n:Nat) =
  ...

let counter () =
  ...

let 2counters =
  let w1 = counter() in
  let w2 = counter() in
  wait (_, _) = sync w1 w2 in
  ()
Goal

\[
\text{letrec count } (w:\text{Widget}) (n:\text{Nat}) = \\
\text{ wait } () = \text{ onClick}(w) \text{ in} \\
\text{ wait } () = \text{ drawButton}(w,n+1) \text{ in} \\
\text{ count } w (n+1)
\]
Goal

```ocaml
letrec count (w:Widget) (n:Nat) =
  wait () = onClick(w) in
  wait () = drawButton(w,n+1) in
  count w (n+1)
```

wait for event

execute

callback

pick a callback

execute callback
letrec count (w:Widget) (n:Nat) =
  wait () = onClick(w) in
  wait () = drawButton(w,n+1) in
  count w (n+1)

Goal
wait for event
execute callback
pick a callback
\( \lambda().k:\text{Data} \rightarrow \bot \)
execute callback
letrec count (w:Widget) (n:Nat) =
  wait () = onClick(w) in
  wait () = drawButton(w,n+1) in
  count w (n+1)
What have we done?

1. What is the langauge for?
2. What features do we need?
3. Define a language
4. Describe how it executes (semantics)
How do we do it?

1. What is the language for?
2. What features do we need?
3. Define a language
4. Describe how it executes (semantics)
How do we do it?

1. What is the language for?
2. What features do we need?
3. Define a language
4. Describe how it executes (semantics)
## Curry-Howard Isomorphism

<table>
<thead>
<tr>
<th>Type System</th>
<th>Type</th>
<th>Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic</td>
<td>Proposition</td>
<td>Proof</td>
</tr>
</tbody>
</table>
Types vs Propositions

\[ A \land B \Rightarrow B \land A \]
Types vs Propositions

\[ A \land B \Rightarrow B \land A \]

\[ \lambda x. (\pi_2 x, \pi_1 x) : A \times B \rightarrow B \times A \]
Terms and Types

\[
\begin{align*}
\Gamma & \vdash x : A \times B \\
\Gamma & \vdash \pi_2 x : B \\
\Gamma & \vdash (\pi_2 x, \pi_1 x) : B \times A \\
\Gamma & \vdash \lambda x.(\pi_2 x, \pi_1 x) : A \times B \to B \times A
\end{align*}
\]
Curry-Howard Isomorphism

Logic $\simeq$ Type System

Properties of Logic $\simeq$ Properties of Programming Languages
<table>
<thead>
<tr>
<th>Logic</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>intuitionistic</td>
<td>pure functional</td>
</tr>
<tr>
<td>classical</td>
<td>callbacks</td>
</tr>
<tr>
<td>temporal</td>
<td>computations</td>
</tr>
<tr>
<td>linear</td>
<td>resource consciousness</td>
</tr>
<tr>
<td>Logic</td>
<td>Feature</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------</td>
</tr>
<tr>
<td>intuitionistic</td>
<td>pure functional</td>
</tr>
<tr>
<td>classical</td>
<td>callbacks</td>
</tr>
<tr>
<td>temporal</td>
<td>computations</td>
</tr>
<tr>
<td>linear</td>
<td>resource consciousness</td>
</tr>
</tbody>
</table>
Classical Logic & Negation

\[ \neg \neg A \equiv A \]

\[ A \rightarrow B \equiv \neg A \lor B \]
Classical Logic & Negation

\[
\neg\neg A \equiv A
\]

\[
A \rightarrow B \equiv \neg A \lor B
\]

\[
A \rightarrow \bot \equiv \neg A \lor \bot \equiv \neg A
\]
Event Loop

wait for event

pick a callback

updateCounter:Data → ⊥

onEvent:(Data → ⊥) → ⊥
Double Negation

\[ A \rightarrow \bot \equiv \neg \neg A \]

\[ \text{onEvent}:(\text{Data} \rightarrow \bot) \rightarrow \bot \]

\[ \neg \neg A \simeq A \]

\[ \text{onEvent}:
eg \neg \text{Data} \]

\[ \text{onEvent}:\text{Data} \]
Double Negation Syntax

```
let x : Data = onEvent \in t
```

\[ \text{onEvent} : \text{Data} \]
Double Negation Syntax

let x : Data = onEvent in t

onEvent : Event

onEvent : (Data \rightarrow \bot) \rightarrow \bot

onEvent (\lambda x : Data . t)
Wait?

\[
\text{wait } x : \text{ Data } = \text{ onEvent in } t
\]
Event Loop

- wait for event
- pick a callback
- updateCounter: Data → ⊥
- execute callback
- onEvent: (Data → ⊥) → ⊥
<table>
<thead>
<tr>
<th>Logic</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>intuitionistic</td>
<td>pure functional</td>
</tr>
<tr>
<td>classical</td>
<td>callbacks</td>
</tr>
<tr>
<td>temporal</td>
<td>computations</td>
</tr>
<tr>
<td>linear</td>
<td>resource</td>
</tr>
<tr>
<td></td>
<td>consciousness</td>
</tr>
</tbody>
</table>
## Temporal Logic

<table>
<thead>
<tr>
<th>Now</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Always</td>
<td>□ A</td>
</tr>
<tr>
<td>Eventually</td>
<td>◊ A</td>
</tr>
</tbody>
</table>
Eventually as Computation
Eventually as Computation

\[
\begin{array}{c}
\Gamma \quad t_1 : \Diamond A \\
\Gamma, x : A \quad t_2 : \Diamond B \\
\hline
\Gamma \quad \text{wait } x = t_1 \text{ in } t_2 : \Diamond B
\end{array}
\]
Double Negation + Time

onEvent: (Data \rightarrow \bot) \rightarrow \bot

onEvent: \Box (Data \rightarrow \bot) \rightarrow \bot
Double Negation + Time

\[ \text{onEvent: } \square (\text{Data} \rightarrow \bot) \rightarrow \bot \]

A \rightarrow \bot \equiv \neg \neg A

\[ \text{onEvent: } \neg \neg \neg \neg \text{Data} \]
Classical Temporal Logic

¬ □ ¬ A

≃

◊ A
Double Negation + Time

\[
\text{onEvent: } \Box (\text{Data} \rightarrow \bot) \rightarrow \bot
\]

\[
A \rightarrow \bot \cong \neg A
\]

\[
\text{onEvent: } \neg \Box \neg \neg \text{Data}
\]

\[
\neg \Box \neg A \cong \Diamond A
\]

\[
\text{onEvent: } \Diamond \text{Data}
\]
Event Loop Syntax

\[
\text{wait } x : \text{Data} = \text{onEvent in t}
\]

wait for event

\[\text{onEvent:◊Data}\]

pick a callback

\[\lambda x. t : \text{Data} \rightarrow \bot\]

execute callback

\[\text{onEvent:}(\text{Data} \rightarrow \bot) \rightarrow \bot\]

\[\text{onEvent}(\lambda x. t)\]
Linear (Time) Temporal Logic

Branching Time

Linear Time
◊ □ A → ◊ □ B → ◊ □ (A ∧ B)
Synchronize

\[ \Gamma \vdash t_1 : \Diamond \Box A \quad \Gamma \vdash t_2 : \Diamond \Box B \]

\[ \Gamma \vdash \text{sync } t_1 \ t_2 : \Diamond \Box (A \times B) \]

let 2counters =
let w1 = counter() in
let w2 = counter() in
wait (_,_) = sync w1 w2 in
()
<table>
<thead>
<tr>
<th>Logic</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>intuitionistic</td>
<td>pure functional</td>
</tr>
<tr>
<td>classical</td>
<td>callbacks</td>
</tr>
<tr>
<td>temporal</td>
<td>computations</td>
</tr>
<tr>
<td>linear</td>
<td>resource consciousness</td>
</tr>
</tbody>
</table>
Features as worlds

Eventually
# Curry-Howard Isomorphism

<table>
<thead>
<tr>
<th>Type System</th>
<th>Type</th>
<th>Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic</td>
<td>Proposition</td>
<td>Proof</td>
</tr>
<tr>
<td>Category</td>
<td>Object</td>
<td>Morphism</td>
</tr>
</tbody>
</table>
"Adjoint functors arise everywhere..."
Current & Future Work

• GUI language
  – localized syntax
  – event loop semantics

• "features as worlds"
  – framework for relationships between worlds