Model-View-Update-Communicate

Session Types meet the Elm Architecture

Simon Fowler
University of Edinburgh

ABCD Final Meeting
19th December 2019

Functional Session Types

$$\mathsf{EqualityClient} : \,!\mathsf{Int}.!\mathsf{Int}.?\mathsf{Bool}.\mathsf{End}$$

$$\begin{array}{l} \mathsf{equalityClient} : \mathsf{EqualityClient} \multimap \mathsf{Bool} \\ \mathsf{equalityClient}(\mathsf{s}) \triangleq \\ \quad \textbf{let}\ \mathsf{s} = \textbf{send}\ (5, \mathsf{s})\ \textbf{in} \\ \quad \textbf{let}\ \mathsf{s} = \textbf{send}\ (5, \mathsf{s})\ \textbf{in} \\ \quad \textbf{let}\ (\mathsf{res}, \mathsf{s}) = \textbf{receive}\ \mathsf{s}\ \textbf{in} \\ \quad \textbf{close}\ \mathsf{s}; \mathsf{res} \end{array}$$

→ Session types: Types for protocols

→ Here, interested in **linear functional languages**

→ Huge advances over the course of ABCD!

Interactivity?

Majority of implementations: Command line applications

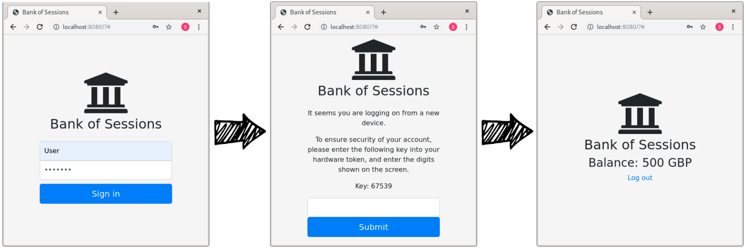
```
[simon@dazzle sessions]$ links calc.links
42 : Int
```

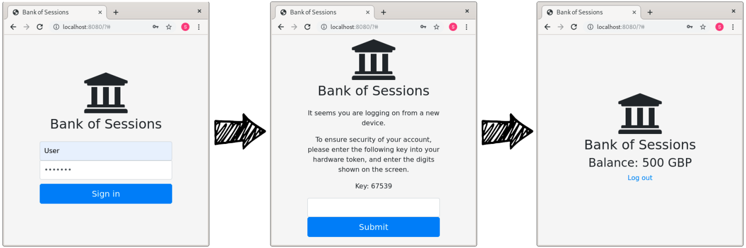
Really, communication actions triggered by UI events, sending user-specified data

Difficult to embed linear resources into a GUI

Some early work on session types + GUIs, but ad-hoc, not formal

→ (Client code in Exceptional Asynchronous Session Types was a **mess**)


Bank of Sessions
localhost:8080/?#
Bank of Sessions
User
Sign in
Bank of Sessions
localhost:8080/?#
Bank of Sessions
It seems you are logging on from a new device.
To ensure security of your account, please enter the following key into your hardware token, and enter the digits shown on the screen.
Key: 67539
Submit
Bank of Sessions
localhost:8080/?#
Bank of Sessions
Balance: 500 GBP
Log out



$$
\begin{aligned}
&\mathsf{TwoFactorClient} \triangleq \\
&\quad !(\mathsf{Username}, \mathsf{Password}).\&\{ \\
&\qquad \mathsf{Authenticated} : \mathsf{ClientBody}, \\
&\qquad \mathsf{Challenge} : ?\mathsf{ChallengeKey}.!\mathsf{Response}.\&\{\mathsf{Authenticated} : \mathsf{ClientBody}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{AccessDenied} : \mathsf{End}\}, \\
&\qquad \mathsf{AccessDenied} : \mathsf{End} \\
&\quad \}
\end{aligned}
$$

Approach

Step 1: Formalise a GUI framework

→ I chose Model-View-Update, as pioneered by Elm

Step 2: Extend formalism with session types

→ Some intricacies...

Step 3: Implement in Links

→ Result: Idiomatic server **and** client code for session-typed web applications

Contributions

λ_{MVU}: A Formal Model of the MVU Architecture

→ First formal characterisation of MVU
→ Soundness proofs

Extending λ_{MVU} with Session Types

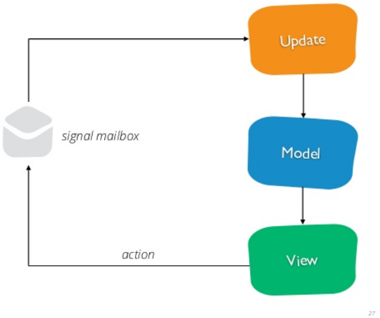
→ Formal characterisations of **subscriptions** and **commands** from Elm
→ **Linearity** and **model transitions** allow safe integration of session types

Implementation and Examples

→ MVU + extensions implemented in Links language
→ Example applications including two-factor authentication and chat server

Demo: A box and a label

Model-View-Update



https://www.slideshare.net/RogerioChaves1/introduction-to-elm

Model: State of application
View: Renders model as HTML
Update: Updates model based on UI messages

Model-View-Update (in Links)

```
typename Model   = (contents: String);
typename Message = [| UpdateBox: String |];

sig view : (Model) ~> HTML(Message)
fun view(model) {
 vdom
  <input
    type="text" value="{model.contents}"
    e:onInput="{fun(str) { UpdateBox(str) }}"/>
  <div>{ textNode(reverse(model.contents)) }</div>
}

sig updt : (Message, Model) ~> Model
fun updt(UpdateBox(newStr), model) {
  (contents = newStr)
}

mvuPage((contents=""), view, updt)
```

Model-View-Update (in Links)

```
typename Model   = (contents: String);
typename Message = [| UpdateBox: String |];

sig view : (Model) ~> HTML(Message)
fun view(model) {
 vdom
  <input
    type="text" value="{model.contents}"
    e:onInput="{fun(str) { UpdateBox(str) }}"/>
  <div>{ textNode(reverse(model.contents)) }</div>
}

sig updt : (Message, Model) ~> Model
fun updt(UpdateBox(newStr), model) {
  (contents = newStr)
}

mvuPage((contents=""), view, updt)
```

Model-View-Update (in Links)

```
typename Model   = (contents: String);
typename Message = [| UpdateBox: String |];

sig view : (Model) ~> HTML(Message)
fun view(model) {
 vdom
  <input
    type="text" value="{model.contents}"
    e:onInput="{fun(str) { UpdateBox(str) }}"/>
  <div>{ textNode(reverse(model.contents)) }</div>
}

sig updt : (Message, Model) ~> Model
fun updt(UpdateBox(newStr), model) {
  (contents = newStr)
}

mvuPage((contents=""), view, updt)
```

Model-View-Update (in Links)

```
typename Model   = (contents: String);
typename Message = [| UpdateBox: String |];

sig view : (Model) ~> HTML(Message)
fun view(model) {
 vdom
  <input
    type="text" value="{model.contents}"
    e:onInput="{fun(str) { UpdateBox(str) }}"/>
  <div>{ textNode(reverse(model.contents)) }</div>
}

sig updt : (Message, Model) ~> Model
fun updt(UpdateBox(newStr), model) {
  (contents = newStr)
}

mvuPage((contents=""), view, updt)
```

λ_{MVU}: Model-View-Update, Formally

Syntax

$$
\begin{array}{llcl}
\text{Types} & A, B, C & ::= & \mathbf{1} \mid A \to B \mid A \times B \mid A + B \mid \mathsf{String} \mid \mathsf{Int} \\
 & & \mid & \mathsf{Html}(A) \mid \mathsf{Attr}(A) \\
\text{String literals} & s & & \\
\text{Integers} & n & & \\
\text{Terms} & L, M, N & ::= & x \mid \lambda x.M \mid M\,N \mid () \mid s \mid n \\
 & & \mid & (M, N) \mid \mathbf{let}\ (x, y) = M\ \mathbf{in}\ N \\
 & & \mid & \mathbf{inl}\ x \mid \mathbf{inr}\ x \mid \mathbf{case}\ L\ \{\mathbf{inl}\ x \mapsto M; \mathbf{inr}\ y \mapsto N\} \\
 & & \mid & \mathbf{htmlTag}\ t\ M\ N \mid \mathbf{htmlText}\ M \mid \mathbf{htmlEmpty} \\
 & & \mid & \mathbf{attr}\ \mathbf{ak}\ M \mid \mathbf{attrEmpty} \mid M \star N
\end{array}
$$

$$
\begin{array}{ll}
\text{Tag names} & t \\
\text{Attribute names} & \mathbf{at}
\end{array}
\qquad
\begin{array}{llcl}
\text{Attribute keys} & \mathbf{ak} & ::= & \mathbf{at} \mid \mathbf{h} \\
\text{Event handler names} & \mathbf{h} & &
\end{array}
$$

Syntactic Sugar

$$
\left[\!\!\left[
\begin{array}{l}
\textbf{html} \\
\quad \text{<input type} = \texttt{"text"} \text{ value} = \{\text{model.contents}\} \\
\qquad \text{onInput} = \{\lambda \text{str.UpdateBox(str)}\}\text{></input>} \\
\quad \text{<div>}\{\textbf{htmlText}\ (\text{reverseString}\ (\text{model.contents}))\}\text{</div>}
\end{array}
\right]\!\!\right]
$$

$$
=
$$

$$
\begin{array}{l}
(\textbf{htmlTag}\ \text{input} \\
\quad ((\textbf{attr}\ \text{type}\ \texttt{"text"}) \star (\textbf{attr}\ \text{value model.contents}) \star \\
\qquad (\textbf{attr}\ \text{onInput}\ (\lambda \text{str.UpdateBox(str)}))) \ \textbf{htmlEmpty}) \star \\
\quad \textbf{htmlTag}\ \text{div}\ \textbf{attrEmpty}\ (\textbf{htmlText}\ \text{reverseString}\ (\text{model.contents}))
\end{array}
$$

Semantics by example: Box and a label

$$\text{model} \triangleq (\text{contents} = \texttt{""})$$

$$\begin{aligned}
&\text{view} \triangleq \lambda \text{model}.\textbf{html} \\
&\quad <\text{input type} = \texttt{"text"} \text{ value} = \{\text{model.contents}\} \\
&\qquad \text{onInput} = \{\lambda \text{str.UpdateBox}(\text{str})\}></\text{input}> \\
&\quad <\text{div}>\{\textbf{htmlText } (\text{reverseString } (\text{model.contents}))\}</\text{div}>
\end{aligned}$$

$$\text{update} \triangleq \lambda \text{UpdateBox}(\text{str}).(\text{contents} = \text{str})$$

Semantics by example: Box and a label

run model view update

Semantics by example: Box and a label

$$\langle (\text{model}, \text{view model}) \mid (\text{view}, \text{update}) \mid \epsilon \rangle ⨟$$

htmlEmpty

Semantics by example: Box and a label

$$\langle (\text{model}, \begin{array}{l} \text{<input type} = \texttt{"text"} \text{ value} = \texttt{""} \\ \quad \text{onInput} = \{\lambda \text{str}.\text{UpdateBox}(\text{str})\}\text{>} \\ \text{</input>} \\ \text{<div></div>} \end{array}) \mid (\text{view}, \text{update}) \mid \epsilon \rangle \fatsemi$$

htmlEmpty

Semantics by example: Box and a label

$\langle \textbf{idle}\ \text{model} \mid (\text{view}, \text{update}) \mid \epsilon \rangle ⨟$

<input type $=$ "`text`" value $=$ ""
  onInput $= \{\lambda \text{str}.\text{UpdateBox}(\text{str})\}$ **@** ϵ></input>
<div **@** ϵ></div>

Semantics by example: Box and a label

$$\langle \textbf{idle}\ \text{model} \mid (\text{view}, \text{update}) \mid \epsilon \rangle ⨾$$

<input type $=$ "text" value $=$ ""
onInput $= \{\lambda \text{str}.\text{UpdateBox}(\text{str})\}$ **@** click$(())\cdot$
keyDown$(75)\cdot$ keyUp$(75)\cdot$ input("k")></input>
<div **@** ϵ></div>

Semantics by example: Box and a label

$$\langle \textbf{idle}\ \mathsf{model} \mid (\mathsf{view}, \mathsf{update}) \mid \epsilon \rangle ⨟$$

```
<input type = "text" value = ""
    onInput = {λstr.UpdateBox(str)} @ input("k")>
</input>
<div @ ε></div>
```

Semantics by example: Box and a label

$$\langle \textbf{idle}\ \mathsf{model} \mid (\mathsf{view}, \mathsf{update}) \mid \epsilon \rangle \parallel (\!(\mathsf{UpdateBox}(\texttt{"k"}))\!) ⨟$$

$$\begin{array}{l}
\texttt{<}\mathsf{input\ type} = \texttt{"text"}\ \mathsf{value} = \texttt{""} \\
\quad \mathsf{onInput} = \{\lambda \mathsf{str}.\mathsf{UpdateBox}(\mathsf{str})\} \\
\ \textbf{@}\ \epsilon\texttt{></}\mathsf{input}\texttt{>} \\
\texttt{<}\mathsf{div}\ \textbf{@}\ \epsilon\texttt{></}\mathsf{div}\texttt{>}
\end{array}$$

Semantics by example: Box and a label

$$\langle \textbf{idle}\ \text{model} \mid (\text{view}, \text{update}) \mid \text{UpdateBox}(\texttt{"k"})\rangle ⨟$$

$$
\begin{array}{l}
\text{<input type} = \texttt{"text"}\ \text{value} = \texttt{""} \\
\quad \text{onInput} = \{\lambda \text{str}.\text{UpdateBox}(\text{str})\} \\
\ \textbf{@}\ \epsilon\text{></input>} \\
\text{<div}\ \textbf{@}\ \epsilon\text{></div>}
\end{array}
$$

Semantics by example: Box and a label

$\langle \text{handle}(\text{model}, (\text{view}, \text{update}), \text{UpdateBox}(\texttt{"k"})) \mid (\text{view}, \text{update}) \mid \epsilon \rangle ⨟$

$$\begin{array}{l} \text{<input type} = \texttt{"text"} \text{ value} = \texttt{""} \\ \quad \text{onInput} = \{\lambda \text{str.UpdateBox}(\text{str})\} \\ \ \textbf{@}\ \epsilon\text{></input>} \\ \text{<div}\ \textbf{@}\ \epsilon\text{></div>} \end{array}$$

$$(\text{where handle}(\text{m}, (\text{v}, \text{u}), \text{msg}) \triangleq \textbf{let } \text{m}' = \text{u}\ (\text{msg}, \text{m}) \textbf{ in } (\text{m}', \text{v m}'))$$

Semantics by example: Box and a label

$$
\left\langle\left(\begin{array}{l}
(\text{contents} = \text{"k"}), \\
\quad \text{<input type} = \texttt{"text"} \text{ value} = \texttt{"k"} \\
\qquad \text{onInput} = \{\lambda \text{str}.\text{UpdateBox}(\text{str})\}\text{>} \\
\quad \text{</input>} \\
\quad \text{<div>k</div>}
\end{array}\right) \;\middle|\; (\text{view}, \text{update}) \;\middle|\; \epsilon \right\rangle \fatsemi
$$

$$
\begin{array}{l}
\text{<input type} = \texttt{"text"} \text{ value} = \texttt{""} \\
\quad \text{onInput} = \{\lambda \text{str}.\text{UpdateBox}(\text{str})\} \\
\;\mathbf{@}\; \epsilon\text{></input>} \\
\text{<div } \mathbf{@}\; \epsilon\text{></div>}
\end{array}
$$

Semantics by example: Box and a label

$\langle \textbf{idle} \;\; (\text{contents} = \texttt{"k"}) \mid (\text{view}, \text{update}) \mid \epsilon \rangle \fatsemi$

$\text{<input type} = \texttt{"text"} \text{ value} = \texttt{"k"}$
$\quad \text{onInput} = \{\lambda \text{str.UpdateBox}(\text{str})\} \; \textbf{@} \; \epsilon\text{></input>}$
$\text{<div} \; \textbf{@} \; \epsilon\text{>k</div>}$

Metatheory

Theorem (Preservation)
If $\Gamma \vdash \mathcal{C}$ and $\mathcal{C} \longrightarrow \mathcal{C}'$, then $\Gamma \vdash \mathcal{C}'$.

Theorem (Event Progress)
If $\cdot \vdash \mathcal{C}$, either:

- there exists some $\mathcal{C}'$ such that $\mathcal{C} \longrightarrow_{\mathsf{E}} \mathcal{C}'$; or
- $\mathcal{C} = \langle \textbf{idle}\ \mathsf{V_m} \mid (\mathsf{V_v}, \mathsf{V_u}) \mid \epsilon \rangle \fatsemi \mathsf{D}$ where D cannot be written $\mathcal{D}[\textbf{htmlTag}_{\vec{\mathsf{e}}}\ \mathsf{t}\ \mathsf{V}\ \mathsf{W}]$ for some non-empty $\vec{\mathsf{e}}$.

Extending λ_{MVU}

Commands

Commands: Allow side effects to be performed by event loop

Example: Asynchronous naïve Fibonacci

Commands

Commands: Allow side effects to be performed by event loop
Example: Asynchronous naïve Fibonacci

$$\text{Model} \triangleq \text{Maybe}(\text{Int}) \qquad \text{Message} \triangleq \text{StartComputation} \mid \text{Result}(\text{Int})$$

Commands

Commands: Allow side effects to be performed by event loop
Example: Asynchronous naïve Fibonacci

$$\text{Model} \triangleq \text{Maybe}(\text{Int}) \qquad \text{Message} \triangleq \text{StartComputation} \mid \text{Result}(\text{Int})$$

```
view : Model → Html(Message)
view = λmodel.html
  {case model {
      Just(result) ↦ htmlText intToString(x);
      Nothing ↦ htmlText "Waiting ..." }   }
  <button onClick = {λ().StartComputation}>Start!</button>
```

Commands

Commands: Allow side effects to be performed by event loop

Example: Asynchronous naïve Fibonacci

$$\text{Model} \triangleq \text{Maybe}(\text{Int}) \qquad \text{Message} \triangleq \text{StartComputation} \mid \text{Result}(\text{Int})$$

$$\begin{array}{l}
\text{view} : \text{Model} \to \text{Html}(\text{Message}) \\
\text{view} = \lambda\text{model}.\textbf{html} \\
\quad \{\textbf{case}\ \text{model}\ \{ \\
\qquad \text{Just}(\text{result}) \mapsto \textbf{htmlText}\ \text{intToString}(x); \\
\qquad \text{Nothing} \mapsto \textbf{htmlText}\ \texttt{"Waiting ..."}\ \} \quad \} \\
\quad \text{<button onClick} = \{\lambda().\text{StartComputation}\}\texttt{>Start!<}/\text{button>}
\end{array}$$

$$\begin{array}{l}
\text{update} : (\text{Message} \times \text{Model}) \to (\text{Model}, \text{Cmd}(\text{Message})) \\
\text{update} = \lambda(\text{msg}, \text{model}). \\
\quad \textbf{case}\ \text{msg}\ \{ \\
\qquad \text{StartComputation} \mapsto (\text{Nothing}, \textbf{cmdSpawn}\ \text{Result}(\text{naïveFib}(1000))) \\
\qquad \text{Result}(x) \mapsto (\text{Just}(x), \textbf{cmdEmpty}) \\
\quad \}
\end{array}$$

Linearity

Stock λ_{MVU} does not support linearity (as m′ is used non-linearly when calculating new model and view):

$$\mathsf{handle}(m, (v, u), msg) \triangleq \textbf{let } m' = u\ m \textbf{ in } ({\color{red}m'}, v\ {\color{red}m'})$$

→ Idea: linear parts of model only used in update, not view.
Extract unrestricted part of the model:

$$\begin{array}{lcl} \mathsf{extract} & : & \mathsf{Model} \to (\mathsf{Model} \times \mathsf{UnrestrictedModel}) \\ \mathsf{view} & : & \mathsf{UnrestrictedModel} \to \mathsf{Html}(\mathsf{Message}) \end{array}$$

$$\begin{array}{l} \mathsf{handle}(m, (v, u, e), msg) \triangleq \textbf{let } m' = u\ (msg, m) \textbf{ in} \\ \qquad \textbf{let } (m', unrM) = e\ m' \textbf{ in} \\ \qquad (m', v\ unrM) \end{array}$$

Demo: PingPong application

PingPong in λ_{MVU}

$$\text{PingPong} \triangleq \mu t.!\text{Ping}.?\text{Pong}.t$$

$$\text{Model} \triangleq \text{Pinging}(\text{PingPong}) \mid \text{Waiting}$$
$$\text{Message} \triangleq \text{Click} \mid \text{Ponged}(\text{PingPong})$$

$$\begin{array}{l} \text{update} \triangleq \lambda(\text{msg}, \text{model}). \\ \quad \textbf{case}\ \text{msg}\ \{ \\ \quad\quad \text{Click} \mapsto \text{handleClick}(\text{model}) \\ \quad\quad \text{Ponged}(c) \mapsto \text{handlePonged}(\text{model}, c) \\ \quad \} \end{array}$$

$$\begin{array}{l} \text{handleClick}(\text{model}) \triangleq \\ \quad \textbf{case}\ \text{model}\ \{ \\ \quad\quad \text{Pinging}(c) \mapsto \\ \quad\quad\quad \textbf{let}\ c = \textbf{send}\ (\text{Ping}, c)\ \textbf{in} \\ \quad\quad\quad \textbf{let}\ \text{cmd} = \\ \quad\quad\quad\quad \textbf{cmdSpawn}\ (\textbf{let}\ (\text{pong}, c) = \textbf{receive}\ c\ \textbf{in} \\ \quad\quad\quad\quad\quad\quad\quad\quad \text{Ponged}(c))\ \textbf{in} \\ \quad\quad\quad (\text{Waiting}, \text{cmd}) \\ \quad\quad \text{Waiting} \mapsto (\text{Waiting}, \textbf{cmdEmpty}) \\ \quad \} \end{array}$$

$$\begin{array}{l} \text{handlePonged}(\text{model}, c) \triangleq \\ \quad \textbf{case}\ \text{model}\ \{ \\ \quad\quad \text{Pinging}(c') \mapsto \\ \quad\quad\quad \textbf{cancel}\ c'; \\ \quad\quad\quad (\text{Pinging}(c), \textbf{cmdEmpty}) \\ \quad\quad \text{Waiting} \mapsto \\ \quad\quad\quad (\text{Pinging}(c), \textbf{cmdEmpty}) \\ \quad \} \end{array}$$

PingPong in λ_{MVU}

$$\text{PingPong} \triangleq \mu t.!\text{Ping}.?\text{Pong}.t$$

$$\begin{aligned}\text{Model} &\triangleq \text{Pinging}(\text{PingPong}) \mid \text{Waiting}\\ \text{Message} &\triangleq \text{Click} \mid \text{Ponged}(\text{PingPong})\end{aligned}$$

$$\begin{array}{l}\text{update} \triangleq \lambda(\text{msg}, \text{model}).\\ \quad \textbf{case}\ \text{msg}\ \{\\ \quad\quad \text{Click} \mapsto \text{handleClick}(\text{model})\\ \quad\quad \text{Ponged}(c) \mapsto \text{handlePonged}(\text{model}, c)\\ \quad \}\end{array}$$

$$\begin{array}{l}\text{handleClick}(\text{model}) \triangleq\\ \quad \textbf{case}\ \text{model}\ \{\\ \quad\quad \text{Pinging}(c) \mapsto\\ \quad\quad\quad \textbf{let}\ c = \textbf{send}\ (\text{Ping}, c)\ \textbf{in}\\ \quad\quad\quad \textbf{let}\ \text{cmd} =\\ \quad\quad\quad\quad \textbf{cmdSpawn}\ (\textbf{let}\ (\text{pong}, c) = \textbf{receive}\ c\ \textbf{in}\\ \quad\quad\quad\quad\quad\quad\quad\quad \text{Ponged}(c))\ \textbf{in}\\ \quad\quad\quad (\text{Waiting}, \text{cmd})\\ \quad\quad \color{red}{\text{Waiting} \mapsto (\text{Waiting}, \textbf{cmdEmpty})}\\ \quad \}\end{array}$$

$$\begin{array}{l}\text{handlePonged}(\text{model}, c) \triangleq\\ \quad \textbf{case}\ \text{model}\ \{\\ \quad\quad \color{red}{\text{Pinging}(c') \mapsto}\\ \quad\quad\quad \color{red}{\textbf{cancel}\ c';}\\ \quad\quad\quad \color{red}{(\text{Pinging}(c), \textbf{cmdEmpty})}\\ \quad\quad \text{Waiting} \mapsto\\ \quad\quad\quad (\text{Pinging}(c), \textbf{cmdEmpty})\\ \quad \}\end{array}$$

Illegal states

Issue

→ Must handle messages impossible in a given state (e.g., receiving a pong while waiting to send a ping)
→ Problem: models treated as sum types

Proposal

→ **Multiple** model types, **transitions** between them
→ Make illegal states unrepresentable!

Model transitions

Waiting state

WModel ≜ Waiting
WUModel ≜ **1**
WMessage ≜ Ponged(c)

wView ≜ λ(). **html**
<button disabled = "true">
Send Ping!
</button>

wUpdate ≜ λ(Ponged(c), Waiting).
transition Pinging(c) pView
pUpdate pExtract **cmdEmpty**

wExtract ≜ λx.(Waiting, ())

Pinging state

PModel ≜ Pinging(PingPong)
PUModel ≜ **1**
PMessage ≜ Click

pView ≜ λ(). **html**
<button onClick = {λ().Click}>
Send Ping!
</button>

pUpdate ≜ λ(Click, Pinging(c)).
let c = **send** (Ping, c) **in**
let cmd =
cmdSpawn (**let** (pong, c) = **receive** c **in**
Ponged(c)) **in**
transition () wView wUpdate wExtract cmd

pExtract ≜ λc.(c, ())

Model transitions

Waiting state

$$\mathrm{WModel} \triangleq \mathrm{Waiting}$$
$$\mathrm{WUModel} \triangleq \mathbf{1}$$
$$\mathrm{WMessage} \triangleq \mathrm{Ponged}(c)$$

```
wView ≜ λ(). html
  <button disabled = "true">
    Send Ping!
  </button>

wUpdate ≜ λ(Ponged(c), Waiting).
  transition Pinging(c) pView
    pUpdate pExtract cmdEmpty

wExtract ≜ λx.(Waiting, ())
```

Pinging state

$$\mathrm{PModel} \triangleq \mathrm{Pinging}(\mathrm{PingPong})$$
$$\mathrm{PUModel} \triangleq \mathbf{1}$$
$$\mathrm{PMessage} \triangleq \mathrm{Click}$$

```
pView ≜ λ(). html
  <button onClick = {λ().Click}>
    Send Ping!
  </button>

pUpdate ≜ λ(Click, Pinging(c)).
  let c = send (Ping, c) in
  let cmd =
    cmdSpawn (let (pong, c) = receive c in
                        Ponged(c)) in
  transition () wView wUpdate wExtract cmd

pExtract ≜ λc.(c, ())
```

Wrapping up

Conclusion

Summary

- → First formal characterisation of MVU architecture
- → First formal integration of session-typed communication and GUI programming
- → Not only Greek: fully implemented in Links, along with examples

Find out more!

- → Draft paper: http://bit.ly/mvu-arxiv
- → Artifact: http://bit.ly/mvu-artifact

@Simon_JF

simon.fowler@ed.ac.uk

http://www.links-lang.org

`opam install links`