Model-View-Update-Communicate

Session Types meet the Elm Architecture

Simon Fowler
University of Edinburgh

ABCD Final Meeting
19th December 2019

THE UNIVERSITY of EDINBURGH
informatics

Functional Session Types

EqualityClient : !Int.!Int.?Bool.End

equalityClient : EqualityClient $\multimap$ Bool
equalityClient(s) $\triangleq$
let s $=$ **send** $(5, \mathrm{s})$ **in**
let s $=$ **send** $(5, \mathrm{s})$ **in**
let $(\mathrm{res}, \mathrm{s}) =$ **receive** s **in**
close s; res

- $\rightarrow$ Session types: Types for protocols
- $\rightarrow$ Here, interested in **linear functional languages**
- $\rightarrow$ Huge advances over the course of ABCD!

Interactivity?

Majority of implementations: Command line applications

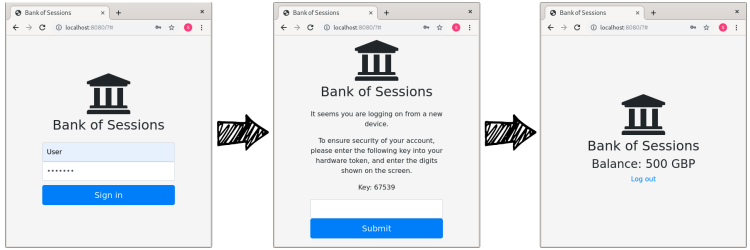
```
[simon@dazzle sessions]$ links calc.links
42 : Int
```

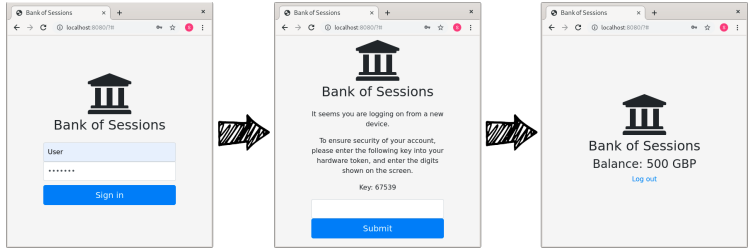
Really, communication actions triggered by UI events, sending user-specified data

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
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
Balance: 500 GBP
Log out



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

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

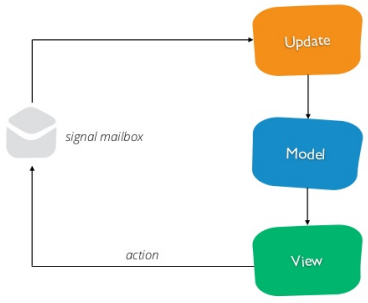
→ Formal characterisations of **subscriptions** and **commands** from Elm
→ **Linearity** and **model transitions** allow safe integration of session types

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

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

Tag names	t		Attribute keys	$\mathbf{ak}$	$::= \mathbf{at} \mid \mathbf{h}$
Attribute names	$\mathbf{at}$		Event handler names	$\mathbf{h}$	

Syntactic Sugar

```
⟦ html
    <input type = "text" value = {model.contents}
      onInput = {λstr.UpdateBox(str)}></input>
    <div>{htmlText (reverseString (model.contents))}</div> ⟧
```

=

```
(htmlTag input
  ((attr type "text") ⋆ (attr value model.contents)⋆
    (attr onInput (λstr.UpdateBox(str)))) htmlEmpty) ⋆
  htmlTag div attrEmpty (htmlText reverseString (model.contents))
```

Semantics by example: Box and a label

model $\triangleq$ (contents = "")

view $\triangleq$ λmodel.**html**
  <input type = "text" value = {model.contents}
    onInput = {λstr.UpdateBox(str)}></input>
  {**htmlText** (reverseString (model.contents))}

update $\triangleq$ λUpdateBox(str).(contents = str)

Semantics by example: Box and a label

run model view update

Semantics by example: Box and a label

$\langle (\text{model}, \text{view model}) \mid (\text{view}, \text{update}) \mid \epsilon \rangle_{\S}^{\S}$

htmlEmpty

Semantics by example: Box and a label

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

htmlEmpty

Semantics by example: Box and a label

$\langle$**idle** model $\mid$ (view, update) $\mid$ $\epsilon\rangle^{\S}_{\S}$

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

Semantics by example: Box and a label

$\langle$**idle** model | (view, update) | $\epsilon\rangle_{\S}$

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

Semantics by example: Box and a label

$\langle$**idle** model $\mid$ (view, update) $\mid$ $\epsilon\rangle^{\circ}_{\circ}$

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

Semantics by example: Box and a label

$$\langle \textbf{idle}\ \text{model} \mid (\text{view}, \text{update}) \mid \epsilon \rangle \parallel ((\text{UpdateBox}(\texttt{"k"})))^{\circ}_{9}$$

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

Semantics by example: Box and a label

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

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

Semantics by example: Box and a label

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

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

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

Semantics by example: Box and a label

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

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

Semantics by example: Box and a label

$\langle$**idle** $(\text{contents} = \texttt{"k"}) \mid (\text{view}, \text{update}) \mid \epsilon\rangle^{\circ}_{\circ}$

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

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 $\mathcal{D}[\textbf{htmlTag}_{\overrightarrow{e}}\ \mathsf{t}\ \mathsf{V}\ \mathsf{W}]$ for some non-empty $\overrightarrow{e}$.

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

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

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

update : (Message × Model) → (Model, Cmd(Message))
update = λmodel.
  case model {
    StartComputation ↦ (Nothing, cmdSpawn Result(naïveFib(1000)))
    Result(x) ↦ (Just(x), cmdEmpty)
  }
```

Linearity

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

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

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

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

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

Demo: PingPong application

PingPong in λ_{MVU}

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

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

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

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

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

PingPong in λ_{MVU}

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

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

```
update ≜ λ(msg, model).
  case msg {
    Click ↦ handleClick(model)
    Ponged(c) ↦ handlePonged(model, c)
  }
```

```
handleClick(model) ≜
  case model {
    Pinging(c) ↦
      let c = send (Ping, c) in
      let cmd =
        cmdSpawn (let (pong, c) = receive c in
                  Ponged(c)) in
      (Waiting, cmd)
    Waiting ↦ (Waiting, cmdEmpty)
  }
```

```
handlePonged(model, c) ≜
  case model {
    Pinging(c′) ↦
      cancel c′;
      (Pinging(c), cmdEmpty)
    Waiting ↦
      (Pinging(c), cmdEmpty)
  }
```

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

WModel $\triangleq$ Waiting
WUModel $\triangleq$ **1**
WMessage $\triangleq$ Ponged(c)

wView $\triangleq \lambda()$. **html**
  <button disabled = "true">
    Send Ping!
  </button>

wUpdate $\triangleq \lambda$(Ponged(c), Waiting).
  transition Pinging(c) pView
    pUpdate pExtract **cmdEmpty**

wExtract $\triangleq \lambda$x.(Waiting, ())

Pinging state

PModel $\triangleq$ Pinging(PingPong)
PUModel $\triangleq$ **1**
PMessage $\triangleq$ Click

pView $\triangleq \lambda()$. **html**
  <button onClick = $\{\lambda()$.Click$\}$>
    Send Ping!
  </button>

pUpdate $\triangleq \lambda$(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 $\triangleq \lambda$c.(c, ())

Wrapping up

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