Model-View-Update-Communicate: Session Types meet the Elm Architecture

Simon Fowler 🌐
University of Edinburgh, Scotland
simon.fowler@ed.ac.uk

Abstract

Session types are a type discipline for communication channel endpoints which allow conformance to protocols to be checked statically. Safely implementing session types requires linearity, usually in the form of a linear type system. Unfortunately, linear typing is difficult to integrate with graphical user interfaces (GUIs), and to date most programs using session types are command line applications.

In this paper, we propose the first principled integration of session typing and GUI development by building upon the Model-View-Update (MVU) architecture, pioneered by the Elm programming language. We introduce $\lambda_{MVU}$, the first formal model of the MVU architecture, and prove it sound. By extending $\lambda_{MVU}$ with commands as found in Elm, along with linearity and model transitions, we show the first formal integration of session typing and GUI programming. We implement our approach in the Links web programming language, and show examples including a two-factor authentication workflow and multi-room chat server.

2012 ACM Subject Classification Software and its engineering → Concurrent programming languages

Keywords and phrases Session types, concurrent programming, Model-View-Update

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.14


Funding This work was supported by ERC Consolidator Grant Skye (grant no. 682315) and an ISCF Metrology Fellowship grant provided by the UK government’s Department for Business, Energy and Industrial Strategy (BEIS).

Acknowledgements I thank Jake Browning for sparking my interest in Elm and for his help with an early prototype of the Links MVU library; Sára Decova for a previous version of the multi-room chat server example; Sam Lindley for many useful discussions and suggestions; and James Cheney, April Gonçalves, and the anonymous ECOOP PC and AEC reviewers for detailed comments.

1 Introduction

Modern applications are necessarily concurrent and distributed. Along with concurrency and distribution naturally comes communication, but communication protocols are typically informally described, resulting in costly runtime failures and code maintainability issues.

Session types [23, 24] are a type discipline for communication channel endpoints which allow conformance to a protocol to checked statically rather than after an application is deployed. Many distributed GUI applications, such as chat applications or multiplayer games, would benefit from session-typed communication with a server. Unfortunately, safely implementing session types requires a require a linear type system, but safely integrating linear resources and GUIs is nontrivial. As a consequence, to date most programs using session types are batch-style applications run on the command line.

The lack of a principled integration of GUI applications and session types is a significant barrier to their adoption. In this paper, we bridge this gap by extending the Model-View-Update (MVU) architecture, pioneered by the Elm programming language, to support linear
resources. We present $\lambda_{MVU}$, a core formalism of the MVU architecture, and an extended version of $\lambda_{MVU}$ which supports session-typed communication. Informed by the formal development, we provide a practical implementation in the Links programming language [10].

**Session types by example.** Let us consider a two-factor authentication workflow, introduced by Fowler et al. [20]. A user first enters their credentials. If correct, the server can then either grant access, or send a challenge key. If challenged, the user enters the challenge code into a hardware token, which generates a response to be entered into the web page. The server then either authenticates the user or denies access.

We can describe the two-factor authentication example as a session type as follows:

```
TwoFactorServer ≜ ?!(Username, Password), ∥{
  Authenticated : ServerBody,
}∥{
  Authenticated : ServerBody,
  AccessDenied : End
}
```

```
TwoFactorClient ≜ !(Username, Password), &{
  Authenticated : ClientBody,
}&{
  Authenticated : ClientBody,
  AccessDenied : End
}
```

The `TwoFactorServer` type shows the session type for the server, which firstly receives (?) the credentials from the client, and then chooses (∥) whether to authenticate, deny access, or issue a challenge. If the server issues a challenge, it sends (!) the challenge string, awaits the response, and then chooses whether to accept or reject the request. The `ServerBody` type abstracts over the actions performed in the remainder of the application, for example taking out a loan. The `TwoFactorClient` type is the dual of the `TwoFactorServer` type: where the server sends, the client receives, and where the client sends, the server receives. The & construct denotes offering a choice of branches. Suppose we have constructs for sending along, receiving from, and closing an endpoint:

```
send : (A × !A.S) → S  receive : ?A.S → (A × S)  close : End → 1
```

Let us also suppose we have constructs for selecting and offering a choice:

```
select $\ell_i$, $M : S_j$  offer $M \{x_i(x_j) \mapsto N_i\}_{i \in I} : A$
```

where $M$ has session type $\oplus \{\ell_i : S_j\}_{i \in I}$, and $j \in I$ where $M$ has session type $\& \{\ell_i : S_j\}_{i \in I}$, each $x_i$ binds an endpoint with session type $S_i$, and each $N_i$ has type $A$

We can write a server implementation as follows:

```
twoFactorServer : TwoFactorServer → 1  
```

```
twoFactorServer(s) ≜ let (username, password), s = receive s in
  if checkDetails(username, password) then
    let s = select Authenticated s in serverBody(s)
  else let s = select AccessDenied s in close s
```

To implement session-typed communication safely, we require a linear type system [44] to ensure each communication endpoint is used exactly once: as an example, without linearity it would be possible to attempt to receive the credentials twice.

**Linearity and GUIs.** We can also write a client application:

```
twoFactorClient : (Username × Password × TwoFactorClient) → 1  
```

```
twoFactorClient((username, password), s) ≜ let s = send ((username, password), s) in
  offer s {Authenticated(s) → clientBody(s)}
  Challenge(s) → let (key, s) = receive s in
    let s = send (generateResponse(key), s) in
    offer s {Authenticated(s) → clientBody(s)}
    AccessDenied(s) → close s; loginFailed
```

```
AccessDenied(s) → close s; loginFailed
```
However, such a client is of little use, as it sends only a pre-defined set of credentials, and the step where a user enters the response to the challenge is replaced by a function `generateResponse`. Ideally, we would like the credentials to be entered into a GUI, and for a button press to trigger the session communication with the server.

Let us attempt to write a GUI for the first stage of the two-factor authentication example; as HTML is well-understood, we concentrate on web pages in the remainder of the paper.

```html
render(c) ≝

＜html＞
＜body＞
＜input id = “username”＞＜/input＞
＜input id = “password”＞＜/input＞
＜button onClick = login(c)＞Submit＜/button＞
＜/body＞
＜/html＞
```

Given a channel `c` of type `TwoFactorClient`, the `render` function generates a web page with input boxes for the username and password, and a button to submit the credentials. The `login` function, triggered when the button is clicked, retrieves the username and password from the two input boxes, and sends the credentials along `c`. The `handleResponse` function, which we omit, receives the response from the server and updates the web page.

On first inspection, this implementation seems sound since the endpoint `c` is used linearly. However, the above attempt is unsound due to the asynchronous nature of GUI programming: there is nothing stopping the user pressing the button twice and sending the credentials twice along `c`, in contravention of the session type. As a further complication, suppose we augmented the protocol with a “forgotten password” branch, triggered by another button. This would require two instances of `c` in the GUI, again violating linearity:

```html
let pid = spawn handler(c) in

＜html＞
＜body＞
＜input id = “username”＞＜/input＞
＜input id = “password”＞＜/input＞
＜button onClick = login(pid)＞Submit＜/button＞
＜/body＞
＜/html＞
```

It is clear that directly embedding linear resources into a GUI is a non-starter. A more successful approach involves spawning a separate process which contains the linear resource, and which receives non-linear messages from the GUI. Upon receiving a GUI message, the process can then perform the session communication, while ignoring duplicate GUI messages:

```html
render(c) ≝

let pid = spawn handler(c) in

＜html＞
＜body＞
＜input id = “username”＞＜/input＞
＜input id = “password”＞＜/input＞
＜button onClick = login(pid)＞Submit＜/button＞
＜/body＞
＜/html＞
```

The `render` function begins by spawning `handler(c)` as a separate process with an incoming message queue (or `mailbox`), returning the process ID `pid`. As before, the `login` function is triggered by pressing the button, and retrieves the credentials from the web page. Instead of communicating on the channel directly, it sends a `SubmitLogin` message containing the credentials to the process ID of handler process, written `pid ! SubmitLogin(user, pass)`. The handler process retrieves the message from its mailbox (`get()`), and can then communicate with the server over the linear endpoint. Such an approach also scales to the “forgotten password” extension, by adding another GUI message.

The above approach is used by Fowler et al. [20], who provide the first integration of session types and web application development, including the ability to gracefully handle failures such as the user closing their browser mid-session. Unfortunately, the approach is
brittle and ad-hoc. All interaction with the web page occurs using imperative operations such as `getContents` and `setContents`; contrary to best practices such as the Model-View-Controller (MVC) [30] pattern, the state of the web page is not derived directly from the data contained by the application. Furthermore, there is no connection between the state of the handler process and what is displayed on the web page: this can easily lead to mismatches between the possible GUI messages which can be sent and which can be handled.

**Model-View-Update.** This paper is about doing better. Our approach is to formalise Model-View-Update, an architectural pattern for GUI development popularised by the Elm programming language [1], and extend it to support linear resources. MVU is an appealing starting point as it is particularly suited to functional programming. Furthermore, MVU has directly inspired popular technologies such as Redux [5] and the Flux architecture [4], which are used with the popular React [2] frontend web framework for JavaScript.

The Elm programming language [1] is a functional programming language designed for writing web applications. Elm was originally designed to use functional reactive programming (FRP) [14], where time-varying signals can be used to construct reactive web applications. A paper describing Elm, and its core formal semantics, was published at PLDI 2013 [12].

For many languages, that would be the end of the story. But unusually for a research language, Elm gained a user community, and a standard architectural pattern known as **The Elm Architecture** grew organically to such a point that Elm abandoned FRP altogether [11]. At its core, The Elm Architecture is a descendant of MVC where a **model** contains the state of the application; a **view** function renders the model; and the rendered model produces messages which are handled by an **update** function to produce a new model. More generally, this pattern has been referred to as **Model-View-Update**, or MVU for short [3, 40].

Consider the following web application, where a user enters text into a text box, and the application displays the text, reversed:

```
We can write this example using MVU as follows:

<table>
<thead>
<tr>
<th>Model</th>
<th>Message</th>
<th>model</th>
<th>update</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model ≜ (contents : String)</td>
<td>Message ≜ UpdateBox(String)</td>
<td>model ≜ (contents = &quot;&quot;)</td>
<td>update ≜ λ(UpdateBox(str), m).{contents = str}</td>
</tr>
</tbody>
</table>

view : Model → Html(Message)  
view ≜ λmodel.html

<input type = "text" value = {model.contents} 
onInput = {λstr.UpdateBox(str)})><input>

{htmlText (reverseString (model.contents))}

</div>

(model, view, update)
```

We define two type aliases: the **Model** captures the state of the application and is defined as a record with a single **String** field, **contents**. **Messages** are produced as a result of user interaction. The **Message** type is defined as a singleton variant type with constructor **UpdateBox**, containing the updated value of the text box.

The **view** function renders a model. It has the type **Model → Html(Message)**, which is a function taking a **Model** as its argument, and returning HTML which may produce messages of type **Message**. The `value = {model.contents}` attribute of the input box states that the contents of the text box should reflect the **contents** field of the model. The `onInput` attribute is an event handler: its body is a function taking the current value of the input box (str) and producing an **UpdateBox** message containing the updated contents of the box. The contents of the div tag are derived from the reversed contents.
Syntax

Types  \( A, B, C ::= 1 \mid A \to B \mid A \times B \mid A + B \mid \text{String} \mid \text{Int} \mid \text{Html}(A) \mid \text{Attr}(A) \)

String literals  \( s \)

Integers  \( n \)

Terms  \( L, M, N ::= x \mid \lambda x.M \mid \text{rec } f(x) . M \mid M N \mid () \mid s \mid n \mid (M, N) \mid \text{let } (x, y) = M \text{ in } N \mid \text{inl } x \mid \text{inr } x \mid \text{case } L \{ \text{inl } x \mapsto M ; \text{inr } y \mapsto N \} \mid \text{htmlTag } t M N \mid \text{htmlText } M \mid \text{htmlEmpty} \mid \text{attr } ak M \mid \text{attrEmpty} \mid M \star N \)

Tag names  \( t \)

Attribute keys  \( ak ::= at \mid h \)

Attribute names  \( at \)

Event handler names  \( h \)

Typing rules for terms

\[
\begin{align*}
\text{T-HTMLTAG:} & \quad \Gamma \vdash M : \text{Attr}(A) \quad \Gamma \vdash N : \text{Html}(A) \quad \Gamma \vdash \text{htmlTag } t M N : \text{Html}(A) \\
\text{T-HTMLTEXT:} & \quad \Gamma \vdash M : \text{String} \quad \Gamma \vdash \text{htmlText } M : \text{Html}(A) \quad \Gamma \vdash \text{htmlEmpty} : \text{Html}(A) \\
\text{T-ATTR:} & \quad \Gamma \vdash M : \text{String} \quad \Gamma \vdash \text{attr } at M : \text{Attr}(A) \quad \Gamma \vdash \text{attrEmpty} : \text{Attr}(A) \\
\text{T-EVTATTR:} & \quad \Gamma \vdash M : \text{ty}(h) \to A \quad \Gamma \vdash \text{attr } h M : \text{Attr}(A) \quad \Gamma \vdash \text{attrEmpty} : \text{Attr}(A) \\
\text{T-HTMLAPPEND:} & \quad \Gamma \vdash M : \text{Html}(A) \quad \Gamma \vdash N : \text{Html}(A) \quad \Gamma \vdash M \star N : \text{Html}(A) \quad \Gamma \vdash M \star N : \text{Attr}(A) \\
\end{align*}
\]

**Figure 1** Syntax and typing rules for \( \lambda_{MVU} \) terms

The update function takes a message and previous model as its arguments, and produces a new model. In this case, the update function constructs a new model where the contents field is set to the payload of the UpdateBox message. Finally, the program is a 3-tuple containing the initial model, and the view and update functions.

To achieve our goal of a formal integration of session typing and GUI programming, we first formalise MVU, and then generalise the architecture to support linear models and messages. Supporting linearity poses some challenges, as we will see in §3.

1.1 Contributions.

The overarching contribution of this paper is the first principled integration of session-typed communication with a GUI framework. Concretely, we make three contributions:

1. We introduce the first formal model of the MVU architecture, \( \lambda_{MVU} \) (§2). We prove (§2.3) that \( \lambda_{MVU} \) satisfies preservation and event progress properties.

2. We extend \( \lambda_{MVU} \) with commands, linearity, and model transitions (§3), which allow \( \lambda_{MVU} \) to support GUIs incorporating session-typed communication, and we prove the soundness of the extended calculus.

3. We implement the architecture in the Links web programming language. We show an extended example of a chat application where client code uses the linear MVU framework, and where client-server communication happens over session-typed channels (§4).

The implementation and examples are available in the paper’s companion artifact.
In this section, we formalise MVU as a core calculus, $\lambda_{MVU}$, an extension of the simply-typed $\lambda$-calculus with products, sums, HTML, and event handling. Even without extensions, $\lambda_{MVU}$ is expressive enough to support many common applications such as form handling.

2 Syntax

Types. Figure 1 shows the syntax and typing rules for $\lambda_{MVU}$. Types are ranged over by $A, B, C$, and consist of the unit type $1$, functions $A \rightarrow B$, products $A \times B$, sums $A + B$, and string and integer types. Types $\text{Html}(A)$ and $\text{Attr}(A)$ are the type of HTML elements and attributes which can produce messages of type $A$.

Terms. Terms, ranged over by $L, M, N$, include variables, $\lambda$ abstractions, anonymous recursive functions, function application, the unit value, string literals, integers, and sum and pair introduction and elimination. The remaining terms encode HTML elements and attributes. The $\text{htmlTag} t M N$ construct denotes an HTML element with tag name $t$ (for example, $\text{div}$), attributes $M$, and children $N$; the $\text{htmlText} M$ construct describes a text node with text $M$; and $\text{htmlEmpty}$ defines an empty HTML node.

The $\text{attr} ak M$ construct describes an attribute with key $ak$ and body $M$, where the key $ak$ is either an attribute name $at$ or an event handler name $h$. The $\text{attrEmpty}$ construct defines an empty attribute.

The $M \star N$ operator appends two HTML elements or attributes. Since both HTML elements and attributes support a unit element ($\text{htmlEmpty}$ and $\text{attrEmpty}$ respectively), elements and attributes together with $\star$ form two monoids.

Events. We model interaction with the Document Object Model (DOM) through events, which model those dispatched by a browser. An event signature is a 3-tuple $(ev, h, A)$ consisting of an event name $ev$, handler name $h$, and payload type $A$. We require a bijective mapping between event and handler names. Figure 2 describes example event signatures used in the remainder of the paper. We consider four primitive events: click, which is fired when an element is clicked; input, which is fired when the contents of a text field are changed; and keyUp and keyDown, which are fired when a key is pressed while focused on an element.

Event handlers are attached to elements as attributes, and generate a message in response to an event. We write $\text{handler}(ev)$ to refer to the handler for $ev$: for example, $\text{handler(click)} = \text{onClick}$. We write $\text{ty}(ev)$ to refer to the payload type of $ev$ and write $\text{ty}(h)$ for the payload type of an event handled by $h$. As an example, both $\text{ty}(\text{click}) = 1$ and $\text{ty}(\text{onClick}) = 1$.

Term typing. Term typing rules for $\lambda$-calculus constructs are standard, so are omitted. Rule $T$-$\text{htmlTag}$ states that $\text{htmlTag} t M N$ can be given type $\text{Html}(A)$ if its attributes $M$ have type $\text{Attr}(A)$ and children have type $\text{Html}(A)$. Text nodes $\text{htmlText} M$ do not produce
Values $U, V, W ::= \lambda x.M \mid \text{rec } f(x). M \mid () \mid (V, W) \mid \text{inl} V \mid \text{inr} V \mid s \mid n$

$::= \text{htmlTag } t V W \mid \text{htmlEmpty} \mid \text{htmlText} V$

$\mid \text{attr } a k V \mid \text{attrEmpty} \mid V * W$

Events $e ::= \text{ev}(V)$

DOM Pages $D ::= \text{domTag}(\varepsilon) t V D \mid \text{domText} V \mid \text{domEmpty} \mid D * D'$

Active thread $T ::= \text{idle } V_0 m \mid M$

Function state $F ::= (V_0, V_0)$

Processes $P, Q ::= \text{run } M \mid (T \mid F) \mid ((M)) \mid P \parallel Q$

Configurations $C ::= P # D$

Process contexts $P ::= [] \mid P \parallel P$

DOM contexts $D ::= [] \mid \text{domTag}(\varepsilon) t V D \mid D * D \mid D * D$

Thread contexts $T ::= \text{run } E \mid (E \mid F) \mid (E \parallel F)$

**Figure 3** Runtime syntax for $\lambda\text{MVU}$

any messages, and so have type Html$(A)$ if $M$ has type String (T-HTML.Text); similarly, $\text{htmlEmpty}$ has type Html$(A)$ (T-HTML.Empty).

Rule T-Attr assigns attributes $\text{attr } at M$ type Attr$(A)$ for any $A$ if $M$ has type String. Rule T-EvtAttr types event handler attributes $\text{attr } h M$: if the event handler $M$ has type $\text{ty}(h) \rightarrow A$ (i.e., it produces messages of type $A$), then the attribute can be given type Attr$(A)$.

Finally, T-AttrEmpty states that the empty attribute $\text{attrEmpty}$ has type Attr$(A)$ for any type $A$. We overload the $*$ operator to append both HTML elements and attributes (T-HTML.Append and T-Attr.Append).

**Syntactic sugar.** We assume the usual encodings of records as pairs and variant types as binary sums, and use pattern matching notation. It is useful to be able to write HTML using XML-like notation, where an antiquoted expression $\{M\}$ allows a term $M$ to be embedded within an HTML tree. The view function from the introduction desugars to:

\[
\lambda\text{model.}
\]

\[
\begin{align*}
\text{(htmlTag input} & \text{ (attr type "text") } \times (\text{attr value model.contents}) \times \\
& (\text{attr onInput (Attr.UpdateBox(str))}) \text{ htmlEmpty} \times \\
& \text{htmlTag div attrEmpty (htmlText reverseString (model.contents))})
\end{align*}
\]

The formal definitions and desugaring translations are unsurprising; the details can be found in the extended version [18].

### 2.2 Operational Semantics

We can now provide $\lambda\text{MVU}$ with a small-step operational semantics.

#### 2.2.1 Runtime Syntax

Figure 3 describes the runtime syntax of $\lambda\text{MVU}$. Values, ranged over by $U, V, W$, are standard. An event $\text{ev}(V)$ consists of event name $\text{ev}$ and payload $V$. We write $\epsilon$ for an empty meta-level sequence, and use $\cdot$ for sequence concatenation. DOM pages, ranged over by $D$, are the runtime representation of HTML, where tags $\text{domTag}(\varepsilon) t V D$ contain an event queue $\varepsilon$ of events dispatched to the element.

**Concurrency.** Concurrency is vital when modelling GUI applications as event handling is asynchronous: computation triggered by a user interaction should not block the UI.
Concurrency is also essential when considering session-typed communication. We therefore formulate the calculus as a concurrent λ-calculus in the style of Niehren et al. [36], by augmenting the simply-typed λ-calculus with processes and concurrent reduction.

**Processes.** An *initialisation process* \( \text{run} M \) evaluates the initial system state written by a user, where \( M \) is a 3-tuple containing the initial model, view function, and update function. An *event loop process* \( \langle T \mid F \rangle \) consists of an active thread \( T \) and function state \( F \) comprising the view and update functions. The thread can either be \( \text{idle} V_m \), meaning the process has current model \( V_m \) and is waiting for another message to process, or evaluating a term \( M \). An *event handler process* \( \langle \langle M \rangle \rangle \) is spawned to generate a message in response to an event.

**Configurations.** Concurrent and event-driven reduction happens in the context of a *system configuration* \( P ; D \), where \( P \) is the concurrent fragment of the system and \( D \) is the current DOM page. An MVU program as written by a user is a term \( M \) specifying the initial model, view function, and update function, of type \( (A \times (A \rightarrow \text{Html}(B))) \times ((B \times A) \rightarrow A) \). A program is evaluated in the context of an *initial configuration*:

> **Definition 1** (Initial configuration). An initial configuration for a term \( M \) is of the form \( \text{run} M \; ; \; \text{domEmpty} \).

**Evaluation contexts.** Term evaluation contexts \( E \) (omitted) are set up for call-by-value, left-to-right evaluation. Process contexts \( \mathcal{P} \) allow reduction under parallel composition. Thread contexts \( \mathcal{T} \) allow reduction inside threads. DOM contexts \( \mathcal{D} \) allow us to focus on each element of a DOM forest; note that they deliberately allow non-unique decomposition in order to support nondeterministic reduction.
2.2.2 Reduction Rules

Figure 4 shows the reduction rules for λMVU processes and configurations; reduction on terms is standard β-reduction. Reduction on configurations is defined modulo the associativity and commutativity of parallel composition.

Differing. As DOM pages include event queues, they contain strictly more information than HTML. To avoid losing pending events, we require a differencing operation. Define \( \text{erase}(D) \) as the operation \( \text{erase}([\text{domTag}(\vec{t}) \ t \ U \ D]) = \text{htmlTag} \ t \ (\text{erase}(D)) \), with the other cases defined recursively. DOM pages can be modified by adding a node with an empty queue, removing a node, or updating a node’s attributes. We define operation \( \text{diff}(U, D) = D' \) if \( \text{erase}(D') = U \), and \( D' \) is obtained from \( D \) by the minimum number of insertions and deletions.

Semantics by example. Let us return to our original example from §1: a box and a text node displaying the reversed box contents. We reuse the \( J \) and \( D \) node, or updating a node’s attributes. We define operation \( \text{erase} \) and \( \text{update} \) by the minimum number of insertions and deletions.

Figure 4 shows the reduction rules for λMVU processes and configurations; reduction on terms is standard β-reduction. Reduction on configurations is defined modulo the associativity and commutativity of parallel composition.

Diffing. As DOM pages include event queues, they contain strictly more information than HTML. To avoid losing pending events, we require a differencing operation. Define \( \text{erase}(D) \) as the operation \( \text{erase}([\text{domTag}(\vec{t}) \ t \ U \ D]) = \text{htmlTag} \ t \ (\text{erase}(D)) \), with the other cases defined recursively. DOM pages can be modified by adding a node with an empty queue, removing a node, or updating a node’s attributes. We define operation \( \text{diff}(U, D) = D' \) if \( \text{erase}(D') = U \), and \( D' \) is obtained from \( D \) by the minimum number of insertions and deletions.

Semantics by example. Let us return to our original example from §1: a box and a text node displaying the reversed box contents. We reuse the \( J \) and \( D \) node, or updating a node’s attributes. We define operation \( \text{erase} \) and \( \text{update} \) by the minimum number of insertions and deletions.

\[
\text{run}(V_m, V_e, V_u) \triangleright \text{domEmpty} \\
\Rightarrow (\text{E-RUN}) \quad (V_m, V_e, V_u) \mid (V_e, V_u) \triangleright \text{domEmpty} \\
\Rightarrow^+ \quad \langle \text{input type = "text" value = ""} \rangle \quad (V_m, \text{onInput} = \lambda \text{str} \cdot \text{UpdateBox}(\text{str}))</input> \mid (V_e, V_u) \triangleright \text{domEmpty} \\
\Rightarrow (\text{E-UPDATE}) \quad \langle \text{input type = "text" value = ""} \rangle \quad (idle \ V_m \mid (V_e, V_u)) \mid \text{onInput} = \lambda \text{str} \cdot \text{UpdateBox}(\text{str})</input> \quad (idle \ V_m \mid (V_e, V_u))
\]

The system now does not reduce until a user interacts with the text box and presses the \( k \) key, modelled by E-INTERACT. At this point, the event queue for the input box receives four events: click, keyDown, keyUp, and input, which are are processed by rule E-EVT. The input element does not have handlers for the click, keyDown, and keyUp events, so no processes are spawned, but \( \text{onInput} \) contains an \( \text{onInput} \) handler, which handles the input event by spawning \( ((\text{UpdateBox}("k")))).

\[
\Rightarrow^+ (\text{E-INTERACT}) \quad \langle \text{input type = "text" value = ""} \rangle \quad \text{onInput} = \lambda \text{str} \cdot \text{UpdateBox}(\text{str})</input> \quad \text{keyDown}(75) \cdot \text{keyUp}(75) \cdot \text{input}("k")</input> \\
\Rightarrow (\text{E-EVT}) \quad \langle \text{input type = "text" value = ""} \rangle \quad \text{onInput} = \lambda \text{str} \cdot \text{UpdateBox}(\text{str})</input> \\
\Rightarrow (\text{E-EVT}) \quad \langle \text{input type = "text" value = ""} \rangle \quad \text{onInput} = \lambda \text{str} \cdot \text{UpdateBox}(\text{str})</input> \\
\Rightarrow (\text{E-EVT}) \quad \langle \text{input type = "text" value = ""} \rangle \quad \text{onInput} = \lambda \text{str} \cdot \text{UpdateBox}(\text{str})</input>
\]

Since UpdateBox("k") is already a value and the event loop process is idle, we can process the message (E-HANDLE). The handle meta-function calculates a new model \( m' \) by applying
### 2.3 Metatheory

**Runtime typing.** To reason about the metatheory, we require runtime typing rules, shown in Figure 5. Judgement $\vdash e$ states that the payload of an event $e$ has the payload type specified by its signature. Judgement $\vdash T : EvtLoop(A, B)$ can be read “Active thread $T$ has model type $A$ and message type $B$”. An idle thread $idle_{V_m}$ has type $EvtLoop(A, B)$ if $V_m$ has type $A$ (TS-IDLE). An active thread $M$ currently processing a message has type $EvtLoop(A, B)$ if $M$ has type $(A \times HTML(B))$, i.e., computes a pair of a new model with type $A$ and HTML which produces messages of type $B$ (TS-PROCESSING).

Judgement $\vdash^* P : A$ states that process $P$ is well typed and produces or consumes messages of type $A$. The parallel composition of two processes $P_1 \parallel P_2$ has message type $A$ if both $P_1$ and $P_2$ have message type $A$ (TP-Par). An event handler process $\langle M \rangle$ has message type $A$ if term $M$ has type $A$ (TP-THREAD).

---

**Table 1:** Runtime typing for $\lambda_{MVU}$

<table>
<thead>
<tr>
<th>Tyeping rules for events $\vdash e$</th>
<th>Typing rules for active threads $\vdash T : EvtLoop(A, B)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{TE-EVT}$ $\vdash V : \text{ty}(v)$</td>
<td>$\text{TS-IDLE}$ $\vdash V_m : A$</td>
</tr>
<tr>
<td>$\vdash ev(V)$</td>
<td>$\vdash idle_{V_m} : EvtLoop(A, B)$</td>
</tr>
<tr>
<td>$\vdash M : (A \times HTML(B)) \times ((B \times A) \rightarrow A)$</td>
<td>$\vdash M : EvtLoop(A, B)$</td>
</tr>
<tr>
<td>$\vdash^* \alpha M \vdash (\langle \alpha \rangle) : A$</td>
<td>$\vdash^* P_1 : A$, $\vdash^* P_2 : A$ $\vdash^* P_1 \parallel P_2 : A$</td>
</tr>
</tbody>
</table>

**Combination of flags**

- $O + O = O$
- $O + * = *$
- $O + * = *$
- $* + * = *$
- $* + * = \text{undefined}$

---

**Figure 5** Runtime typing for $\lambda_{MVU}$

The update function to a pair of the previous model and the message, calculates a new HTML value $v'$ by applying the view function to $m'$, and returns the pair $(m', v')$. Finally, the page is diffed against the previous DOM page to generate a new DOM page $D'$, and the event loop process reverts to being idle:

$\rightarrow (E-P-HANDLE)$

$(\lambda \epsilon) M \vdash (\langle \lambda \epsilon \rangle) : \text{A}$

$\langle \text{onInput} = \{\lambda \text{str}. \text{UpdateBox}(\text{str})\} \rangle |
\langle (\lambda \epsilon) \rangle$
An initialisation process \( \text{run} M \) is well-typed if \( M \) is a product type where each component has the correct model, view, and update types. An event loop process \( \langle T \mid (V_v, V_u) \rangle \) has message type \( B \) if its active thread \( T \) has model type \( A \) and message type \( B \); its view function \( V_v \) has type \( A \rightarrow \text{Html}(B) \); and its update function has type \( (B \times A) \rightarrow A \) (TP-EVENTLOOP).

Thread flags \( \phi \) ensure that there is precisely one initialisation process or event loop process in a process typeable under flag \( \bullet \).

Judgement \( \vdash C \) states that configuration \( C \) is well-typed: a system configuration \( P ; D \) is well-typed if process \( P \) has precisely one event loop process with message type \( A \) and page \( D \) has type \( \text{Page}(A) \). The omitted typing rules for pages (of shape \( \vdash D : \text{Page}(A) \)) are similar to those for terms of type \( \text{Html}(A) \).

Note that we consider only closed configurations and processes since it makes little sense for DOM pages \( D \) to contain free variables, and because processes do not bind variables.

We are now well-placed to state some formal results. We omit proofs in the main body of the paper; full proofs can be found in the extended version [18].

**Preservation.** Reduction preserves typing.

\[ \text{Theorem 2 (Preservation). If } \vdash C \text{ and } C \rightarrow C', \text{ then } \vdash C'. \]

**Progress.** The system vacuously satisfies a progress property as it can always reduce by E-INTERACT due to user input. It is more interesting to consider the event progress property enjoyed by the system without E-INTERACT: either there are no events to process and the system is idle, or the system can reduce. Functional reduction satisfies progress.

\[ \text{Lemma 3 (Progress (Terms)). If } \cdot \vdash M : A, \text{ then either } M \text{ is a value, or there exists some } N \text{ such that } M \rightarrow_M N. \]

Let \( \rightarrow_E \) be the relation \( \rightarrow \) without rule E-INTERACT. The concurrent fragment of the language will reduce until all event handler threads have finished evaluating, and there are no more messages to process. By appeal to Lemma 3, we can show event progress.

\[ \text{Theorem 4 (Event Progress). If } \vdash C, \text{ either:} 
\begin{enumerate}
  \item there exists some \( C' \) such that \( C \rightarrow_E C' \); or 
  \item \( C = \langle \text{idle} V_m \mid (V_v, V_u) \rangle ; D \) where \( D \) cannot be written \( D[\text{domTag}(\overrightarrow{v}) \ t V W] \) for some non-empty \( \overrightarrow{v} \).
\end{enumerate} \]

3 \( \lambda_{MVU} \) with Session Types

In this section, we extend \( \lambda_{MVU} \) to support session types. We require three extensions: commands, to perform side-effects; linearity, to implement session types safely; and transitions, to allow multiple model and message types. We begin by showing each extension by example, and show the extended formalism in §3.4.

3.1 Commands

Real-world applications require side-effects. To this end, Elm supports commands which describe side-effects to be performed in the event loop. Although commands in Elm are more general, for our purposes, it is particularly useful to be able to spawn a process which will run concurrently and eventually return a message. As an example, we may want to await the result of an expensive computation, and display the result when the computation completes.
Letting naïveFib\(x\) be the naïve Fibonacci function and assuming an \texttt{intToString} function, we can write:

\[
\begin{align*}
\text{Model} &\triangleq \text{Maybe}(\text{Int}) & \text{Message} &\triangleq \text{StartComputation} \mid \text{Result}(\text{Int}) \\
\text{view} : \text{Model} &\rightarrow \text{Html}(\text{Message}) & \text{view} &= \lambda \text{model} \cdot \text{html} \\
\text{html} &\begin{cases}
\text{case model} \{ \\
\quad \text{Just}(\text{result}) \rightarrow \text{htmlText} \ \text{intToString}(\text{result}); \\
\quad \text{Nothing} \rightarrow \text{htmlText} \ \text{“Waiting...”} \}
\end{cases} \\
\text{html} &\end{align*}
\]

The model is of type \text{Maybe}(\text{Int}), with value \text{Just}(V) for some integer value \(V\) if the result has been computed, or \text{Nothing} if the application is awaiting the result. The \text{Message} type is a variant type consisting of \text{StartComputation} which is sent to start the computation, and \text{Result}(\text{Int}), which is sent to return a result. The \text{view} function renders either the result, or “Waiting...” if no result is available.

3.2 Linearity

As we showed in §1, safely implementing session types requires linearity: we therefore require linear model and message types. Linearity would also prove useful for other linear resources such as functional arrays with in-place update [44]. Unfortunately, \(\lambda_{\text{MVU}}\) as defined so far does not support linearity. Consider \text{handle}:

\[
\begin{align*}
\text{handle}(m, (v, u), \text{msg}) &\triangleq \text{let } \text{m}' = u(\text{msg}, m) \text{ in } (\text{m}', v \ \text{m}')
\end{align*}
\]

The updated model, \(\text{m}'\), is used non-linearly as it is returned for use in subsequent requests, and also used to render the model as HTML.

**Extraction.** Linear resources are needed only when updating the model—not when rendering the webpage—as we will not need to communicate on session channels when rendering. If the developer implements a function:

\[
\text{extract} : A \rightarrow (A \times B)
\]

where \(A\) is the type of a model, and \(B\) is the unrestricted fragment of the model, we can restore linear usage of the model (letting \(e\) be the extraction function):

\[
\begin{align*}
\text{handle}(m, (v, u, e), \text{msg}) &\triangleq \text{let } \text{m}' = u(\text{msg}, m) \text{ in } \\
&\text{let } (\text{m}', \text{unrM}) = e \ \text{m}' \text{ in } (\text{m}', v \ \text{unrM})
\end{align*}
\]
S. Fowler 14:13

\[
\text{Model } \triangleq (\text{Bool} \times \text{Chan}(\text{Ping}) \times \text{Chan}(\text{Pong}))
\]

\[
\text{Message } \triangleq \text{Click} \mid \text{Ponged}
\]

\[
\text{update} : (\text{Message} \times \text{Model}) \rightarrow \text{Model}
\]

\[
\text{update } \triangleq \lambda (\text{msg}, (\_ , \text{pingCh}, \text{pongCh})).
\]

\[
\text{case msg } \{
\]

\[
\text{Click } \rightarrow \text{let cmd } = \text{cmdSpawn (send(Ping, pingCh);}
\]

\[
\text{let Pong } = \text{receive pongCh in}
\]

\[
\text{Ponged in ((false, pingCh, pongCh), cmd)}
\]

\[
\text{Ponged } \rightarrow \{(\text{true, pingCh, pongCh}, \text{cmdEmpty})
\]

[Figure 6 PingPong application using simply-typed channels]

An alternative approach would be to assign the view function type \(A \rightarrow (A \times \text{Html}(B))\), returning the linear model and allowing it to be re-bound. We would need to modify \(\text{handle}\):

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

A key disadvantage of this approach is that rendering is no longer a read-only operation, breaking an important abstraction barrier.

**Example.** We can now write our first session-typed \(\text{\lambdaMVU}\) application. Our web client consists of a button which, when clicked, triggers the sending of a \text{Ping} message to the server. Once clicked, the button is disabled. The server then receives the \text{Ping} message and responds with a \text{Pong} message; upon receiving the response, the client then re-enables the button.

**Simply-typed channels.** Before considering a session-typed version of the application, it is instructive to consider a version without session typing, shown in Figure 6. Let \(\text{Chan}(A)\) be the type of a simply-typed channel over which one can send and receive values of type \(A\). The model is a 3-tuple containing a Boolean value which is true when waiting for the user to click the “Send Ping!” button, and false when waiting for a response; a channel for \text{Ping} messages; and a channel for \text{Pong} messages. There are two types of UI message: \text{Click} denotes that the button has been clicked, and \text{Ponged} denotes that a \text{Pong} message has been received along the \text{Pong} channel.

The \text{view} function displays the page, adding the \text{disabled} attribute to the button if we are waiting for a \text{Pong} message. The \text{update} function case-splits on the UI message: in the case of a \text{Click} message raised by the button, the model is updated to set the \text{pinging} flag to false, and the function creates a command to send a \text{Ping} message along \text{pingCh}, receive a \text{Pong} message from \text{pongCh}, and return a \text{Ponged} UI message. In the case of a \text{Ponged} message, the model is updated to set the \text{pinging} flag to true, enabling the button again. The \text{server} function models a server thread, which repeatedly receives \text{Ping} messages from \text{pingCh} and sends \text{Pong} messages to \text{pongCh}. 
Model-View-Update-Communicate

Session types. Session types $S$ range over output $!A.S$, input $?A.S$, the completed session $\text{End}$, recursive session types $\mu.S$, and (possibly dualised) recursive type variables $t$. We take an equi-recursive treatment of recursive session types, identifying a recursive session type with its unfolding. We omit types and constructs for branching and selection as they can be encoded [13, 28]. The $\text{send}$ constant sends a value of type $A$ over an endpoint of type $!A.S$ and returns the continuation of the session, $S$. The $\text{close}$ constant closes a completed session endpoint. The $\text{receive}$ constant takes an endpoint of type $?A.S$ and receives a pair of a value of type $A$ and endpoint of type $S$. The $\text{cancel}$ constant allows an endpoint to be discarded safely [20, 35].

$$\text{Session types} \quad S \quad ::= \quad !A.S \mid ?A.S \mid \mu.S \mid t \mid \text{End}$$

$$\text{send} : (A \times !A.S) \rightarrow S \quad \text{receive} : ?A.S \rightarrow (A \times S) \quad \text{close} : \text{End} \rightarrow 1 \quad \text{cancel} : S \rightarrow 1$$

Figure 7 shows the PingPong client written in $\lambda_{MVU}$. We can encode the PingPong protocol as a session type, $\mu!(\text{Ping}.)?\text{Ping}.t$. The $\text{Model}$ type encodes the two states of the application: $\text{Pinging}(c)$ is the state where the “Send Ping!” button is enabled and the user can send a Ping message along session channel $c$, whereas $\text{Waiting}$ is the state where the button is disabled and awaiting a Pong message from the other party. The $\text{UModel}$ type is the unrestricted model type which does not include the session channel. Again, the $\text{Message}$
type encodes the UI message in the application: the Click UI message is produced when the button is pressed, whereas the Ponged(PingPong) UI message is produced when a Pong session message has been received. Note that the Ponged UI message now contains a session channel of type PingPong as a parameter.

The view function takes an unrestricted model and displays a button, which is disabled in the Waiting state but enabled in the Pinging state. The extract function pairs the linear model with the associated unrestricted model.

The update function case-splits on the message. The handleClick function handles the Click message, and case-splits on the model. If the model is in the Pinging(c) state, then the function creates a command to spawn a process which will send a Ping message along c, receive a Pong message along c, and generate a Ponged UI message when the Pong message is received. The function finally updates the model to the Waiting state. If the model is in the Waiting state—which should not occur, since the button is disabled—then the model remains the same and no command is created.

The handlePonged function handles a Ponged(c) message. Again, we must case split on the model. If the model is in the Waiting state, then we can change to the Pinging state, given endpoint c. However, if the model is in the Pinging(c’ state and a Ponged message is received—which should not occur, since according to the session type, there is no way for the peer to send a Pong message while we are waiting to send a Ping—we now have two linear resources. We choose to discard c’ using cancel, and change the model to Pinging(c’), but this is an arbitrary choice to satisfy a code path that must exist, but should never be used.

3.3 Model transitions

Our proposal is still not quite satisfactory: as we saw with the PingPong example, we need to include cases in the update function which cannot arise. We highlight these in red. This is even more pronounced when dealing with linear resources, such as needing to handle a Ponged message when waiting to send a Ping.

\[
\begin{align*}
\text{Ponged state} & \\
PModel & = \text{Pinging}(\text{PingPong}) \\
\text{PUModel} & = \text{1} \\
PMessage & = \text{Click} \\
pView & : \text{PUModel} \rightarrow \text{Html}(\text{PMessage}) \\
pView & = \lambda \cdot \text{html} \\
\text{html} & <\text{body}> \\
& <\text{button onClick} = \{ \lambda !. \text{Click} \} > \text{Send Ping}! \\
& <\text{/body}> \\
\text{pUpdate} & : (\text{PMessage} \times \text{PModel}) \rightarrow \\
& \text{Transition}(\text{PModel}, \text{PMessage}) \\
pUpdate & = \lambda (\text{Click}, \text{Pinging}(c)). \\
\text{let cmd} & = \text{cmdSpawn} ( \text{let c = send} (\text{Ping}, c) \text{ in} \\
& \text{let (pong, c) = receive} c \text{ in} \text{Ponged}(c)) \text{ in} \\
\text{transition} & \text{Waiting wView wUpdate wExtract cmd} \\
pExtract & : \text{PModel} \rightarrow (\text{PModel} \times \text{PUModel}) \\
pExtract & = \lambda x. (x, ())
\end{align*}
\]

\[
\begin{align*}
\text{Waiting state} & \\
\text{WModel} & = \text{Waiting} \\
\text{WUModel} & = \text{1} \\
\text{WMessage} & = \text{Ponged}(c) \\
wView & : \text{WUModel} \rightarrow \text{Html}(\text{WMessage}) \\
wView & = \lambda \cdot \text{html} \\
\text{html} & <\text{body}> \\
& <\text{button disabled = "true"} > \text{Send Ping!} \\
& <\text{/body}> \\
\text{wUpdate} & : (\text{WMessage} \times \text{WModel}) \rightarrow \\
& \text{Transition}(\text{WModel}, \text{WMessage}) \\
wUpdate & = \lambda (\text{Ponged}(c), \text{Waiting}). \\
\text{transition} & \text{Pinging}(c) \text{ pView} \\
pUpdate & \text{pExtract \text{cmdEmpty}} \\
wExtract & : \text{WModel} \rightarrow \\
& (\text{WModel} \times \text{WUModel}) \\
wExtract & = \lambda x. (x, ())
\end{align*}
\]

Figure 8 PingPong application using model transitions
Kinds $\kappa ::= L \mid U$

Types $A, B, C ::= \mathbf{1} \mid A \rightarrow^\kappa B \mid A \times B \mid A + B \mid \text{String} \mid \text{Int} \mid S \mid \text{Html}(A) \mid \text{Attr}(A) \mid \text{Cmd}(A) \mid \text{Transition}(A, B)$

Session types $S ::= \! A.S \mid ?A.S \mid \!\!S \mid t \mid \top \mid \text{End}$

Terms $L, M, N ::= x \mid \lambda x. M \mid M N \mid K M \mid () \mid s \mid n \mid (M, N) \mid \text{let} \ (x, y) = M \text{ in } N \mid \text{htmlTag} t M N \mid \text{htmlText} M \mid \text{attr} \text{ ak} M \mid \text{attrEmpty} \mid \text{cmdSpawn} M \mid \text{cmdEmpty} \mid M * N \mid \text{transition} M_m M_a M_e M_r \mid \text{noTransition} M_m M_e \mid \text{raise} \mid \text{try} L \text{ as } x \text{ in } M \text{ otherwise } N$

Constants $K ::= \text{send} \mid \text{receive} \mid \text{new} \mid \text{cancel} \mid \text{close}$

Figure 9 Syntax of extended calculus

The problem is that we are encoding the Model type using a sum type, whereas in fact we require multiple model types, and a way to transition between them.

Example. Figure 8 shows how we can modify PingPong to use multiple model types. The left-hand side of the figure shows the Pinging state: the model type consists of the singleton variant tag Pinging(PingPong) containing an endpoint of type PingPong, the unrestricted model is the unit type, and the only message that the Pinging state can receive is Click. The pView function is similar to before, and the pExtract function returns a pair of the current state and the unit value. The pUpdate function is more interesting. Given the current state and a Click message, the function constructs a command which will send the Ping session message, receive the Pong session message, and then generate a Ponged(c) UI message containing the session channel. The function transitions into the Waiting state using the transition primitive, which allows the developer to specify new model, view, update, extract functions, and a command to evaluate. The functions for the Waiting state follow a similar pattern. Session types rule out the communication errors besetting the example in Figure 6, and model transitions eliminate the redundant code paths arising due to illegal states.

3.4 λMVU with Commands, Linearity, and Transitions

Commands, linearity, and transitions are the three key ingredients needed to extend MVU to support models which include session-typed channels. In this section, we introduce a calculus which combines all three extensions, and prove that the extended calculus is sound.

3.4.1 Syntax and Typing

Figure 9 shows the syntax of $\lambda_{MVU}$ extended with commands, linearity, and transitions.

Types and kinds. To support linearity, types are assigned kinds, ranged over by $\kappa$. Types can either be linear (L) or unrestricted (U). A value of linear type must be used precisely once, whereas a value of unrestricted type can be used many times.

We modify function types to include a kind annotation: linear functions may close over linear variables and so must be used once. To support commands, we introduce type $\text{Cmd}(A)$ which is the type of a command which produces messages of type $A$. To support transitions, we introduce type $\text{Transition}(A, B)$ which is parameterised by the current model type $A$ and message type $B$. Finally, we extend types to include session types $S$ as described in §3.2.
Terms. Term \texttt{cmdSpawn} \( M \) is a command which can spawn term \( M \) as a thread, and is monoidally composable using \( \ast \) and \texttt{cmdEmpty}.

There are two terms for transitions: the noTransition \( M_m, M_c \) term denotes that no transition is to occur, and that the model should be updated to \( M_m \) and command \( M_c \) should be evaluated; and transition \( M_m, M_a, M_u, M_e, M_c \) denotes that a transition should occur, with new model \( M_m \), view function \( M_v \), update function \( M_u \), extraction function \( M_e \), and command \( M_c \) to be run once the transition has taken place.

To support session typing, we introduce session typing constants, ranged over by \( K \). We also introduce an application form for constants, \( KM \).

Finally, as discussed in §3.2, it is useful to be able to explicitly discard (or cancel) a session channel. In particular, cancellation is crucial in order to handle the interplay between linearity and transitions, as all unprocessed messages (which may contain linear resources) must be safely discarded when a transition occurs.

Following Mostrous and Vasconcelos [35] and Exceptional GV (EGV) by Fowler et al. [20], if a thread tries to receive from an endpoint whose peer has been cancelled, an exception is raised (raise). Exceptions can be handled using the \( \text{try} \space L \space \text{as} \space x \space \text{in} \space M \space \text{otherwise} \space N \) construct, which tries to evaluate term \( L \), and binds the result to \( x \) in \( M \) if the term evaluates to a value, and evaluates \( N \) if the term raises an exception.

Kinding and subkinding. The kinding relation \( A :: \kappa \) assigns kind \( \kappa \) to type \( A \); our formulation is inspired by that of Padovani [38]. Base types and HTML and attribute types are unrestricted. The kind of a function type is determined by its kind annotation. Session types are linear. The kinds of product, sum, command and transition types are determined by the kinds of their type parameters. The reflexive subkinding rule \( U \leq L \) combined with the kinding subsumption rule states that if a value can be used many times, then it can also be treated as only being used once. We write \( \Gamma :: \kappa \) if \( A :: \kappa \) for each \( x : A \in \Gamma \).

\begin{itemize}
  \item \textbf{Definition 5 (Kinding and subkinding).} We define the subkinding relation as the reflexive relation on kinds \( \leq \) such that \( U \leq L \). We define the kinding relation \( A :: \kappa \) as the largest relation between types and kinds such that:
  \begin{itemize}
    \item \( A :: \kappa' \) if \( A :: \kappa \) and \( \kappa \leq \kappa' \)
    \item \( S :: L \)
    \item \( A :: U \) if \( A \in \{1, \text{String, Int, Html}(B), \text{Attr}(B)\} \)
    \item \( A \rightarrow^\kappa B :: \kappa \)
    \item \( \text{Cmd}(A) :: \kappa \) if \( A :: \kappa \)
    \item \( C :: \kappa \) if \( C \in \{A \times B, A + B, \text{Transition}(A,B)\} \) and both \( A :: \kappa \) and \( B :: \kappa \)
  \end{itemize}
\end{itemize}

Term typing. Figure 10 shows the typing rules for the extended calculus. The splitting relation \( \Gamma = \Gamma_1 + \Gamma_2 \) [8] splits a typing context \( \Gamma \) into two subcontexts which may share only unrestricted variables. We support linearity by changing T-VAR to only type a variable in an unrestricted context, and by using the context splitting judgement when typing subterms. The adaptation of the remaining rules to use context splitting is standard, so we omit them.

The constant application rule \( T\text{-AppK} \) types term \( KM \) and makes use of the type schema function \( \Sigma(K) \) to ensure that the argument \( M \) is of the correct type. Rule \( T\text{-CmdSpawn} \) assigns term \texttt{cmdSpawn} \( M \) type \texttt{Cmd}(\( A \)) if term \( M \) has type \( A \), and rules \( T\text{-CmdEmpty} \) and \( T\text{-CmdAppend} \) allow commands to be composed monoidally.

Rule \( T\text{-Transition} \) types a transition term. The typing rule ensures that the types of the new model, and view, update and extract functions are compatible. Note that the type parameters of the Transition\( (A', B') \) need not match the types of the new model and
functions. Rule T-NoTransition assigns term noTransition $M N$ type Transition($A, B$) if new model $M$ has type $A$, and $N$ is a command of type $Cmd(B)$. Note that in this way, the noTransition $M N$ term replaces the standard result of the update function.

Rule T-Try types an exception handler: the continuations share a typing environment, but the success continuation is augmented with the a variable of the type of the possibly-failing continuation. Finally, raise can have any type as is it does not return (T-Raise).

The type and kinding system ensures that the kind of type $A$ determines the kind of the typing environment needed to type a term of type $A$.

> **Lemma 6.** If $\Gamma \vdash M : A$ and $A :: \kappa$, then $\Gamma :: \kappa$.

**Duality.** The duality relation for session types is standard: output types are dual to input types; we use a self-dual End type; and we use the formulation of the duality of recursive session types advocated by Lindley and Morris [32].

### 3.4.2 Operational Semantics

**Runtime syntax.** Figure 11 shows the runtime syntax for the combined calculus. We introduce runtime names $c, d$ which identify session channel endpoints.
A process for the to spawn is given as an argument to introduce server processes be executed upon updating the page for example, the displayed after the transition. Each state records values which are required in later states: extracting the state to render the new HTML. If there is a model transition, then the thread moves to the transitioning state machine based on whether a model transition occurs. The updating a state machine to model a process running on the server; the thread denoted by an endpoint c, denotes an endpoint c, following the transitioning state to calculate the new HTML to be displayed after the transition. Each state records values which are required in later states: for example, the rendering state records the new model Vm and the command to be executed upon updating the page Vc.

We introduce four new types of process. To model client-server communication, we introduce server processes [M] to model a process M running on the server; the thread to spawn is given as an argument to run. As an example, we could write a Ponger server process for the PingPong example, which immediately responds with a Pong message:

```plaintext
let (c, s) = new () in
(Ping(c), pView, pUpdate, pExtract, cmdEmpty, ponger(s))
ponger(s) = λ().
(rec f(s).
  let (Ping, s) = receive s in
  let s = send (Pong, s in f s) s
```

A name restriction (ved)P binds runtime names c and d in process P, following the double-binder formulation due to Vasconcelos [43]. A zapper thread \(\xi c\) denotes an endpoint c.

---

**Figure 11** Runtime syntax for extended calculus

The biggest departure is that we require a richer structure on active threads, which form a state machine based on whether a model transition occurs. The idle state is as before, and the updating state evaluates the update function. If there is no model transition, then the thread moves to the extracting state to extract the unrestricted model, and the rendering state to render the new HTML. If there is a model transition, then the thread moves to the extractingT state followed by the transitioning state to calculate the new HTML to be displayed after the transition. Each state records values which are required in later states: for example, the rendering state records the new model Vm and the command to be executed upon updating the page Vc.

```
let (c, s) = new () in
(Ping(c), pView, pUpdate, pExtract, cmdEmpty, ponger(s))
ponger(s) = λ().
(rec f(s).
  let (Ping, s) = receive s in
  let s = send (Pong, s in f s) s
```

A name restriction (ved)P binds runtime names c and d in process P, following the double-binder formulation due to Vasconcelos [43]. A zapper thread \(\xi c\) denotes an endpoint c.
that has been cancelled and cannot be used in future communications; we write $\xi V$ to mean $\xi c_1 \| \cdots \| \xi c_n$ for $c_i \in \text{fn}(V)$, where $\text{fn}(V)$ enumerates the free runtime names in a value $V$, and extend this sugar to evaluation contexts. The $\text{halt}$ process denotes that the event loop process has terminated due to an unhandled exception.

We extend evaluation contexts in the standard way, and introduce a class of pure contexts $E_p$, which are evaluation contexts which do not contain any exception handling frames.

**Versions.** Versions $\iota$ ensure that threads spawned in a previous state do not deliver incompatible messages. We annotate event loop processes and event handler threads with versions: given an event loop $\langle T \mid F \rangle_\iota$, a thread $\langle \langle M \rangle \rangle_\iota$, where $\iota \neq \iota'$ can be of arbitrary type as it will be discarded. We write $\text{version}(P) = \iota$ if $P$ contains a subprocess $\langle T \mid F \rangle_\iota$.

**Reduction.** Figures 12 and 13 show the extended process equivalence and reduction rules. Rule E-Try handles evaluation of the success continuation of an exception handler, and
Reduction of configurations

\[ C \rightarrow C' \]

\[ \text{E-RUN} \]
\[ \mathcal{P} \text{[run}(V_m, V_c, V_u, V_e, \lambda(), \mathcal{M})\text{]} \parallel D \rightarrow \text{(extracting}[V_c] (V_e, V_m) \mid (V_e, V_u, V_c) \parallel \parallel [\mathcal{M}] \parallel D \]

\[ \text{E-UPDATE} \]
\[ \mathcal{P} \text{[rendering}[V'_m, V_c] U \mid F'_c] \parallel D \rightarrow \mathcal{P} \text{[idle} V'_m \mid F'_c \parallel \parallel [\mathcal{M}_1], c \parallel \cdots \parallel [\mathcal{M}_n], c] \parallel D' \]
where \( \text{diff}(U, D) = D' \) and \( \text{procs}(V_c) = \overrightarrow{M} \)

\[ \text{E-TRANSITION} \]
\[ \mathcal{P} \text{[transitioning}[V_m, F', V_c] U \mid F'_c] \parallel D \rightarrow \mathcal{P} \text{[idle} V_m \mid F', c \parallel \cdots \parallel [\mathcal{M}_n], c] \parallel D' \]
where \( i = i + 1 \), \( \text{diff}(U, D) = D' \)

\[ \text{E-EVT} \]
\[ P \parallel D \text{[domTag}(\text{ev}(W), \cdot \overrightarrow{c}) \text{]} t U D] \rightarrow P \parallel \cdots \parallel (V_n W), \parallel \cdots \parallel (V_m W), \parallel P \parallel D \text{[domTag}(\overrightarrow{c}) \text{]} t U D] \]
where \( \text{handlers}(\text{ev}, U) = \overrightarrow{V} \) and \( \text{version}(P) = i \)

(Figure 13 Reduction rules for extended calculus (2))

Cancellation of pure active thread contexts

\[ \text{\#updating} \ E_p = \#E_p \]
\[ \text{\#rendering}[V_m, V_c] E_p = \#V_m \parallel \#V_c \parallel \#E_p \]
\[ \text{\#extracting}[V_c] E_p = \#V_c \parallel \#E_p \]
\[ \text{\#transitioning}[V_m, F, V_c] E_p = \#V_m \parallel \#V_c \parallel \#E_p \]

(MVU reduction. MVU reduction rules are specific to MVU. Central to safely integrating
linearity and transitions are rules E-DISCARD, E-DISCARDHALT, and E-HANDLE. Rule
E-HANDLE is modified so that the event loop process only handles a message if the message
has the same version. If the versions do not match, then E-DISCARD safely discards any
channel endpoints in the discarded message by generating zapper threads. Rules E-EXTRACT,
E-RENDER, and E-RENDERT handle the state machine transitions described in
Figure 11 and are used to calculate the new model and HTML.

Session reduction. Session reduction rules encode session-typed communication and are mostly standard: E-New generates a name restriction and returns two fresh endpoints;
E-COMM handles synchronous communication; and E-CLOSE discards the endpoints of a
completed session. The remaining session communication rules handle session cancellation,
and are a synchronous variant of Exceptional GV described by Fowler et al. [20]. Rule
E-CANCEL discards an endpoint. Rules E-SENDZAP, E-RECVZAP, and E-CLOSEZAP raise
an exception if a thread tries to communicate along an endpoint whose peer is cancelled,
ensuring resources are discarded safely.

Exception extraction. Rule E-RAISEH describes exception handling: as raise occurs in a
pure context, the exception is handled by the innermost handler; the rule evaluates the failure
continuation and discards all linear resources in the aborted context. Rules E-RAISEURUN
and E-RAISEUMAIN apply to unhandled exceptions in a main thread, generating the
We modify T-Name and TT-Rendering to unrestricted. The rules for active threads ensure that the types of the terms being evaluated.

Figure 14 shows the runtime typing rules for the extended calculus. Rule T-N-NAME types channel endpoints, and rule TE-EVT mandates that event payload types are unrestricted. The rules for active threads ensure that the types of the terms being evaluated.

halt configuration and cancelling any linear resources in the aborted context. Rules E-RAISETHREAD and E-RAISEUSERSERVER apply to unhandled exceptions in event loop thread and server threads respectively, by cancelling any channels in the aborted continuation.

Configuration reduction. Figure 13 shows the modified configuration reduction rules. We modify E-RUN to take into account the new arguments, and spawn the given server thread. We modify E-UPDATE to spawn threads described by the returned command; E-TRANSITION is similar but changes the function state and increments the version. We modify E-EVT to tag each spawned event handler thread with the version of the event handler process.

3.4.3 Metatheory

Runtime typing. Figure 14 shows the runtime typing rules for the extended calculus. Rule T-N-NAME types channel endpoints, and rule TE-EVT mandates that event payload types are unrestricted. The rules for active threads ensure that the types of the terms being evaluated.
correspond to the state in the state machine (for example, that the updating state returns a term of type Transition(A, B)), and that any recorded values have the correct types.

Let Ψ range over environments containing only runtime names: Ψ := · | Ψ, c : S. We write Ψ₁, Ψ₂ for the disjoint union of environments Ψ₁ and Ψ₂.

We modify the shape of the process typing judgement to Ψ ⊢_φ P : A, which can be read “under typing environment Ψ and thread flag φ, process P has type A and version υ”.

We modify rule TP-EVENTLOOP to include the extraction function, and mandate that the unrestricted model type C has kind U. We modify rule T-THREAD to state that type of an event handler thread (⟨M⟩), has type A if term M has type A and the version matches that of the event handler process. Rule TP-OLDTHREAD allows a thread to have a mismatching type to the event handler process if the versions are incompatible. Finally, TP-ZAP and TP-HALT type zapper threads and the halt thread, and TP-Nu types a name restriction (νcd)P by adding c and d with dual session types into the typing environment.

Properties. The extended calculus satisfies preservation.

▶ Theorem 7 (Preservation). If ⊢ C and C → C′, then ⊢ C′.

Although session types rule out deadlock within a single session, without imposing a tree-like structure on processes [31, 45] (which is too inflexible for our purposes) or using techniques such as channel priorities [29, 37, 39], it is not possible to rule out deadlocks when considering multiple sessions. Since communication over multiple sessions can introduce deadlocks, we begin by proving an error-freedom property, similar to that of Gay and Vasconcelos [21]. An error process involves a communication mismatch.

▶ Definition 8 (Error process). A process P is an error process it contains one of the following processes as a subprocess:
1. (νcd)(T[send(V, c)] || T′[send(W, d)])
2. (νcd)(T[send(V, c)] || T′[close d])
3. (νcd)(T[receive c] || T′[receive d])
4. (νcd)(T[receive c] || T′[close d])

Configuration typing ensures error-freedom.

▶ Theorem 9 (Error-freedom). If Ψ ⊢_φ P : A, then P is not an error process.

Error-freedom shows that session typing ensures the absence of communication mismatches. What remains is to show that, apart from the possibility of deadlock, the additional features do not interfere with the progress property enjoyed by λMVU. We begin by classifying the notion of a blocked thread, which is a thread blocked on a communication action.

▶ Definition 10 (Blocked thread). We say a thread T[M] is blocked if either M = send(V, W), M = receive V, or M = close V.

Let us refer to halt, ⟨T | F⟩₁, and run M as main threads, and (⟨M⟩₁, [M], and ξc as auxiliary threads. Each well-typed configuration has precisely one main thread.

We can now classify the notion of progress enjoyed by the extended calculus. Either the configuration can reduce; is waiting for an event; has halted due to an unhandled exception; or is deadlocked. Again, let →₁ be the →₁ relation without E-INTERACT.

▶ Theorem 11 (Weak Event Progress). Suppose ⊢ C. Either there exists some C′ such that C → C′, or there exists some C′ such that C ≡ C′ and:
Client session types

typename ClientConnect = ?(RoomName).ClientSelect;
typename ClientSelect = !(RoomName, Nickname).
[&| JoinedOK: ?(Topic, [Nickname], ClientReceive) . ClientSend,
JoinedOKAsModerator: ?(Topic, [Nickname], ClientReceive, ModeratorSend). ClientSend,
Nope: ?ConnectError.End [&];

ClientReceive = [&|
IncomingChatMessage: ?(Nickname, Message). ClientReceive,
NewUser: ?(Nickname). ClientReceive,
NewTopic: ?(Topic). ClientReceive,
UserLeft: ?(Nickname). ClientReceive,
UserMuted: ?(Nickname). ClientReceive,
UserUnmuted: ?(Nickname). ClientReceive,
BecomeModerator: ?ModeratorSend. ClientReceive,
Kick: End [&];

ClientSend = [+|
ChatMessage: !(Message).ClientSend,
ChangeTopic: !(Topic).ClientSend,
Leaving: End [+];

ModeratorSend = [+|
KickUser: !(Nickname).ModeratorSend,
MuteUser: !(Nickname).ModeratorSend,
MakeModerator: !(Nickname).ModeratorSend
[+];

Figure 15 Chat server application

1. $D$ cannot be written $D[\text{domTag}(\overrightarrow{e})] tV D$ for a non-empty $\overrightarrow{e}$.
2. If the main thread of $C'$ is halt, then all auxiliary threads are blocked or zapper threads.
3. If the main thread of $C'$ is run $M$, then $M$ is blocked, and all auxiliary threads are either blocked, values, or zapper threads.
4. If the main thread of $C'$ is $(T \mid F)_\iota$, then:
   a. if $T = \text{idle}_M$, then each auxiliary thread is either blocked or a zapper thread; or
   b. if $T = \sum_{L}$, then $L$ is blocked, and each auxiliary thread is either blocked, a value, or a zapper thread.

4 Implementation and Example Application

We have implemented an MVU library for the Links tierless web programming language, which includes all extensions in the paper; Links already has a linear type system and distributed session types, so is an ideal fit.

We now describe a chat application, extending the application presented by Fowler et al. [20]. The application (Figure 15) has two main stages shown to the user: on the first, the user is presented with a list of rooms, and enters a username and selects a room. If a user with the given nickname is not already in the selected room, then the user joins the room, receiving the current topic, a list of other nicknames, and a channel used to receive messages from the server. The user can then send chat messages, change the topic, and leave the
room. If the user is the first user in the room, then they join as a moderator and receive an additional channel which can be used to kick, mute, or promote other users to moderators. Users can receive incoming chat messages, and system messages detailing changes such as a new topic or a user joining the room.

We can encode these interaction patterns using session types. Links session type notation for offering a choice is \([&|...|&]\), and making a choice is \([+|...|+]\). Type `ClientConnect` describes the client receiving the room list. Type `ClientSelect` describes the client sending the room name and nickname, and receiving the response from the server: either joining as a regular user (JoinedOK); joining as a moderator (JoinedOKAsModerator); or an error (Nope). Types `ClientSend` and `ClientReceive` detail the messages that the client can send to, and receive from the server, respectively. Type `ModeratorSend` details privileged moderator actions.

Although the original version of Links [10] ran as a CGI script, modern Links applications run as a persistent webserver. Upon execution, the chat application creates an access point for sessions of type `ClientConnect`, which supports session establishment, and spawns an acceptor thread to accept incoming requests on the access point. Each chat room is represented as a process on the server. When an HTTP request is made, the response contains the MVU application and the access point ID which can be used to establish a session of type `ClientConnect`. After the initial HTTP response, further communication between the client and server happens over a WebSocket [16].

The application has three states: connection, chatting, and a “waiting” state shown while waiting for a response. For the purposes of the paper, we consider the connection state.

The `NotConnectedModel` is the unrestricted part of the model, and contains the current nickname (`nickname`), list of rooms (`rooms`), selected room (`selectedRoom`), value of the “new room” text box (`newRoomText`), and an optional error message to display (`error`). The model, `NCModel`, is a pair of a session endpoint of type `ClientSelect` and a `NotConnectedModel`. The UI messages are described by the `NCMessage` type: for example, the `UpdateNickname` message is generated by the `onInput` event of the nickname input box.

Upon receiving the `SubmitJoinRoom` UI message when the form is submitted, the application can send the nickname and selected room along the `ClientSelect` channel, all of which are contained in the model, without requiring ad-hoc messaging or imperative updates.

5 Related work

Flapjax [34] was the first web programming language to use functional reactive programming (FRP) [14] in the setting of web applications. Flapjax provides `behaviours`, which are variables whose contents change over time, and `event streams`, which are an infinite stream of discrete events which change a behaviour. ScalaLoci [46] is a multi-tier reactive programming framework written in Scala, where changes in reactive `signals` are propagated across tiers, rather than using explicit message passing. Ur/Web [9] and WebSharper UI [19] store data in mutable variables, and allow views of the data to be combined using monadic combinators.

Felleisen et al. [15] describe an earlier approach similar to MVU written in the DrScheme [17] system. Similar to the MVU update function, events such as key presses and mouse movements are handled using functions of type \((\text{Model} \times \text{Event}) \rightarrow \text{Model}\). The approach
handles “environment” events rather than events dispatched by individual elements, and the approach is not formalised. Environment events can be handled using subscriptions in Elm, which can be added to λMVU (see the extended version of the paper [18]).

React [2] is a popular JavaScript UI framework. In React, a user defines data models and rendering functions, and similar to Elm, updates are propagated to the DOM by diffing. Differently to MVU, there is no notion of a message, and a page consists of multiple components rather than being derived from a single model. We expect some technical machinery from λMVU (e.g., event queues, DOM contexts, and diffing) could be reused when formalising React. Redux [5] is a state container for JavaScript applications: to modify the state, one dispatches an action, and a function takes the previous state and an action and produces a new state. In combination with React, the approach strongly resembles MVU.

Hop.js [41] is a multi-tier web framework written in JavaScript. Hop.js services allow remote function invocation, and the framework supports client-side message-passing concurrency using Web Workers [22], but there is no cross-tier message-passing concurrency.

Session types were introduced by Honda [23] and were first considered in a linear functional language by Gay and Vasconcelos [21]; Wadler [45] later introduced a session-typed functional language GV and a logically-grounded session-typed calculus CP (following Caires and Pfenning [7]), and translated GV into CP. Lindley and Morris [31] introduced an operational semantics for GV, and showed type- and semantics-preserving translations between GV and CP. GV inspires FST [33], which is the core calculus for Links’ treatment of session typing.

Fowler et al. [20] extend GV with failure handling, and extend Links with cross-tier session-typed communication. They do not formally consider GUI development, and their approach to frontend web programming using session types (described in Section 1) leads to a disconnect between the state of the page and the application logic. We build upon their approach to session-typed web programming, while also allowing idiomatic GUI development.

King et al. [27] present a toolchain for writing web applications which respect multiparty session types [25]. Protocols are compiled to PureScript [42] using a parameterised monad [6] to guarantee linearity, and the authors integrate their encoding of session types with the Concur UI framework [26]. Each application may only have a single session connecting the client and server, whereas in our system there may be multiple; our approach supports first-class linearity and cross-tier typechecking; our approach is formalised; and our approach supports failure handling. Links does not yet support multiparty session types.

6 Conclusion

Session types allow conformance to protocols to be checked statically. The last few years have seen a flurry of activity in implementing session types in a multitude of programming languages, but linearity—a vital prerequisite for implementing session types safely—is difficult to reconcile with the asynchronous nature of graphical user interfaces. Consequently, the vast majority of implementations using session types are command line applications, and the few implementations which do integrate session types and GUIs do so in an ad-hoc manner.

In this paper, we have addressed this problem by extending the Model-View-Update architecture, pioneered by the Elm programming language. We have presented the first formal study of MVU by introducing a core calculus, λMVU. Leveraging our formal characterisation of MVU, we have introduced three extensions: commands, linearity, and model transitions, enabling us to present the first formal integration of session-typed communication with a GUI framework. Informed by our formal model, we have implemented our approach in Links. As future work, we will investigate how to encode allowed transitions as a behavioural type.
References

14:28 REFERENCES