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

Simon Fowler[1]
University of Edinburgh, United Kingdom
simon.fowler@glasgow.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 λ_{MVU}, the first formal model of the MVU architecture, and prove it sound. By extending λ_{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

Related Version An extended version of the paper is available on arXiv (`https://arxiv.org/abs/1910.11108`).

Supplementary Material ECOOP 2020 Artifact Evaluation approved artifact available at `https://doi.org/10.4230/DARTS.6.2.13`.

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.

[1] now at University of Glasgow, United Kingdom

34th European Conference on Object-Oriented Programming (ECOOP 2020).
Editors: Robert Hirschfeld and Tobias Pape; Article No. 14; pp. 14:1–14:28

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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 λ_{MVU}, a core formalism of the MVU architecture, and an extended version of λ_{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 ≜                          TwoFactorClient ≜
  ?(Username, Password).⊕{                   !(Username, Password).&{
    Authenticated : ServerBody,                Authenticated : ClientBody,
    Challenge : !ChallengeKey.?Response.       Challenge : ?ChallengeKey.!Response.
      ⊕{Authenticated : ServerBody,              &{Authenticated : ClientBody,
        AccessDenied : End},                       AccessDenied : End},
    AccessDenied : End}                        AccessDenied : End}
```

The TwoFactorServer type shows the session type for the server, which firstly receives (?) the credentials from the client, and then chooses ($\oplus$) 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:

$$\textbf{send} : (A \times !A.S) \to S \qquad \textbf{receive} : ?A.S \to (A \times S) \qquad \textbf{close} : \mathsf{End} \to \mathbf{1}$$

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

$\textbf{select}\ \ell_j\ M : S_j$	where M has session type $\oplus\{\ell_i : S_i\}_{i \in I}$, and $j \in I$
$\textbf{offer}\ M\ \{\ell_i(x_i) \mapsto N_i\}_{i \in I} : A$	where M has session type $\&\{\ell_i : S_i\}_{i \in I}$, 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.

```
render(c) ≜
  <html>
    <body>
      <input id = "username"></input>
      <input id = "password"></input>
      <button onClick = login(c)>Submit</button>
    </body>
  </html>
```

```
login(c) ≜ λ().
  let user = getContents("username") in
  let pass = getContents("password") in
  let c = send ((user, pass), c) in
  handleResponse(c)
```

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:

```
<button onClick = login(c)>Submit</button>
<button onClick = reset(c)>Reset password</button>
```

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:

```
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>
```

```
login(pid) ≜ λ().
  let user = getContents("username") in
  let pass = getContents("password") in
  pid ! SubmitLogin(user, pass)

handler(c) ≜
  case (get ()) {
    SubmitLogin(user, pass) ↦
      let c = send ((user, pass), c) in
      handleResponse(c)
  }
```

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:

```
Model ≜ (contents : String)
Message ≜ UpdateBox(String)

model : Model
model ≜ (contents = "")

update : (Message × Model) → Model
update ≜ λ(UpdateBox(str), m).(contents = str)
```

```
view : Model → Html(Message)
view ≜ λmodel.html
  <input type = "text" value = {model.contents}
    onInput = {λstr.UpdateBox(str)}></input>
  <div>
    {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 $A, B, C ::= \mathbf{1} \mid A \to B \mid A \times B \mid A + B \mid \mathsf{String} \mid \mathsf{Int} \mid \mathsf{Html}(A) \mid \mathsf{Attr}(A)$

String literals s

Integers n

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

Tag names t — Attribute keys $ak ::= at \mid h$

Attribute names at — Event handler names h

Typing rules for terms $\boxed{\Gamma \vdash M : A}$

T-HtmlTag
$$\frac{\Gamma \vdash M : \mathsf{Attr}(A) \qquad \Gamma \vdash N : \mathsf{Html}(A)}{\Gamma \vdash \mathbf{htmlTag}\,\mathsf{t}\,M\,N : \mathsf{Html}(A)}$$

T-HtmlText
$$\frac{\Gamma \vdash M : \mathsf{String}}{\Gamma \vdash \mathbf{htmlText}\,M : \mathsf{Html}(A)}$$

T-HtmlEmpty
$$\frac{}{\Gamma \vdash \mathbf{htmlEmpty} : \mathsf{Html}(A)}$$

T-Attr
$$\frac{\Gamma \vdash M : \mathsf{String}}{\Gamma \vdash \mathbf{attr}\,at\,M : \mathsf{Attr}(A)}$$

T-EvtAttr
$$\frac{\Gamma \vdash M : \mathsf{ty}(h) \to A}{\Gamma \vdash \mathbf{attr}\,h\,M : \mathsf{Attr}(A)}$$

T-AttrEmpty
$$\frac{}{\Gamma \vdash \mathbf{attrEmpty} : \mathsf{Attr}(A)}$$

T-HtmlAppend
$$\frac{\Gamma \vdash M : \mathsf{Html}(A) \qquad \Gamma \vdash N : \mathsf{Html}(A)}{\Gamma \vdash M \star N : \mathsf{Html}(A)}$$

T-AttrAppend
$$\frac{\Gamma \vdash M : \mathsf{Attr}(A) \qquad \Gamma \vdash N : \mathsf{Attr}(A)}{\Gamma \vdash M \star N : \mathsf{Attr}(A)}$$

Figure 1 Syntax and typing rules for λ_{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, λ_{MVU} (§2). We prove (§2.3) that λ_{MVU} satisfies preservation and event progress properties.
2. We extend λ_{MVU} with *commands*, *linearity*, and *model transitions* (§3), which allow λ_{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.

Event name ev	Event Handler h (handler(ev))	Payload type (ty(ev), ty(h))	Payload Description
click	onClick	**1**	Unit value
input	onInput	String	Updated contents of a text field
keyUp	onKeyUp	Int	Key code
keyDown	onKeyDown	Int	Key code

Figure 2 Example event signatures.

2 Model-View-Update, Formally

In this section, we formalise MVU as a core calculus, λ_{MVU}, an extension of the simply-typed λ-calculus with products, sums, HTML, and event handling. Even without extensions, λ_{MVU} is expressive enough to support many common applications such as form handling.

2.1 Syntax

Types. Figure 1 shows the syntax and typing rules for λ_{MVU}. Types are ranged over by A, B, C, and consist of the unit type $\mathbf{1}$, functions $A \to B$, products $A \times B$, sums $A + B$, and string and integer types. Types $\mathsf{Html}(A)$ and $\mathsf{Attr}(A)$ 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, λ 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 **htmlTag** t M N construct denotes an HTML element with tag name t (for example, `div`), attributes M, and children N; the **htmlText** M construct describes a text node with text M; and **htmlEmpty** defines an empty HTML node.

The **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 **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 (**htmlEmpty** and **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 handler(ev) to refer to the handler for ev: for example, handler(click) = onClick. We write ty(ev) to refer to the payload type of ev and write ty(h) for the payload type of an event handled by h. As an example, both ty(click) = **1** and ty(onClick) = **1**.

Term typing. Term typing rules for λ-calculus constructs are standard, so are omitted. Rule T-HtmlTag states that **htmlTag** t M N can be given type $\mathsf{Html}(A)$ if its attributes M have type $\mathsf{Attr}(A)$ and children have type $\mathsf{Html}(A)$. Text nodes **htmlText** M do not produce any messages, and so have type $\mathsf{Html}(A)$ if M has type String (T-HtmlText); similarly, **htmlEmpty** has type $\mathsf{Html}(A)$ (T-HtmlEmpty).

$$
\begin{array}{lrcl}
\text{Values} & U, V, W & ::= & \lambda x.M \mid \mathbf{rec}\, f(x)\,.\, M \mid () \mid (V, W) \mid \mathbf{inl}\, V \mid \mathbf{inr}\, V \mid s \mid n \\
& & \mid & \mathbf{htmlTag}\ \mathsf{t}\ V\ W \mid \mathbf{htmlEmpty} \mid \mathbf{htmlText}\ V \\
& & \mid & \mathbf{attr}\ ak\ V \mid \mathbf{attrEmpty} \mid V \star W \\
\text{Events} & e & ::= & \mathsf{ev}(V) \\
\text{DOM Pages} & D & ::= & \mathbf{domTag}(\overrightarrow{e})\ \mathsf{t}\ V\ D \mid \mathbf{domText}\ V \mid \mathbf{domEmpty} \mid D \star D' \\
\text{Active thread} & T & ::= & \mathbf{idle}\ V_m \mid M \\
\text{Function state} & F & ::= & (V_v, V_u) \\
\text{Processes} & P, Q & ::= & \mathbf{run}\ M \mid \langle T \mid F \rangle \mid (\!(M)\!) \mid P \parallel Q \\
\text{Configurations} & \mathcal{C} & ::= & P \fatsemi D \\
\\
\text{Process contexts} & \mathcal{P} & ::= & [\,] \mid \mathcal{P} \parallel P \\
\text{DOM contexts} & \mathcal{D} & ::= & [\,] \mid \mathbf{domTag}(\overrightarrow{e})\ \mathsf{t}\ V\ \mathcal{D} \mid \mathcal{D} \star D \mid D \star \mathcal{D} \\
\text{Thread contexts} & \mathcal{T} & ::= & \mathbf{run}\ E \mid \langle E \mid F \rangle \mid (\!(E)\!)
\end{array}
$$

Figure 3 Runtime syntax for λ_{MVU}.

Rule T-Attr assigns attributes **attr** *at* M type $\mathsf{Attr}(A)$ for any A if M has type String. Rule T-EvtAttr types event handler attributes **attr** $h\ M$: if the event handler M has type $\mathsf{ty}(h) \to A$ (i.e., it *produces messages of type* A), then the attribute can be given type $\mathsf{Attr}(A)$. Finally, T-AttrEmpty states that the empty attribute **attrEmpty** has type $\mathsf{Attr}(A)$ for any type A. We overload the $\star$ operator to append both HTML elements and attributes (T-HtmlAppend and T-AttrAppend).

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:

λ*model*.
  (**htmlTag** input
    ((**attr** type "text") $\star$ (**attr** value *model.contents*)$\star$
      (**attr** onInput (λ*str*.UpdateBox(*str*)))) **htmlEmpty**) $\star$
    htmlTag div **attrEmpty** (**htmlText** reverseString (*model.contents*))

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 λ_{MVU} with a small-step operational semantics.

2.2.1 Runtime Syntax

Figure 3 describes the runtime syntax of λ_{MVU}. Values, ranged over by U, V, W, are standard. An event $\mathsf{ev}(V)$ consists of event name ev and payload V. We write ϵ 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 $\mathbf{domTag}(\overrightarrow{e})\ \mathsf{t}\ V\ D$ contain an event queue $\overrightarrow{e}$ 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.

Meta-level definitions

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

$$\begin{aligned}
\mathsf{handlers}(\mathsf{ev}, \textbf{attrEmpty}) &= \epsilon \\
\mathsf{handlers}(\mathsf{ev}, V \star W) &= \mathsf{handlers}(\mathsf{ev}, V) \cdot \mathsf{handlers}(\mathsf{ev}, W) \\
\mathsf{handlers}(\mathsf{ev}, \textbf{attr}\ at\ V) &= \epsilon \\
\mathsf{handlers}(\mathsf{ev}, \textbf{attr}\ h\ V) &= \begin{cases} V & \text{if } \mathsf{handler}(\mathsf{ev}) = h \\ \epsilon & \text{otherwise} \end{cases}
\end{aligned}$$

Process reduction $\boxed{P \longrightarrow P'}$

$$\begin{aligned}
&\text{EP-HANDLE} && \langle \textbf{idle}\ V_m \mid F\rangle \parallel (\!(V)\!) \longrightarrow \langle \mathsf{handle}(V_m, F, V) \mid F\rangle \\
&\text{EP-PAR} && P_1 \parallel P_2 \longrightarrow P_1' \parallel P_2 \quad \text{if } P_1 \longrightarrow P_1' \\
&\text{EP-LIFTT} && \mathcal{T}[M] \longrightarrow \mathcal{T}[N] \quad \text{if } M \longrightarrow_{\mathsf{M}} N
\end{aligned}$$

Configuration reduction $\boxed{\mathcal{C} \longrightarrow \mathcal{C}'}$

$$\begin{aligned}
&\text{E-RUN} && \mathcal{P}[\textbf{run}\ (V_m, V_v, V_u)] \fatsemi D \longrightarrow \mathcal{P}[\langle (V_m, V_v\ V_m) \mid (V_v, V_u)\rangle] \fatsemi D \\
&\text{E-UPDATE} && \mathcal{P}[\langle (V_m, U) \mid F\rangle] \fatsemi D \longrightarrow \mathcal{P}[\langle \textbf{idle}\ V_m \mid F\rangle] \fatsemi D' \quad \text{where } \mathsf{diff}(U, D) = D' \\
&\text{E-INTERACT} && P \fatsemi \mathcal{D}[\textbf{domTag}(\overrightarrow{e})\ \mathsf{t}\ U\ D] \longrightarrow P \fatsemi \mathcal{D}[\textbf{domTag}(\overrightarrow{e} \cdot \mathsf{ev}(V))\ \mathsf{t}\ U\ D] \\
& && \qquad \text{for some } \mathsf{ev}, V \text{ such that } \vdash \mathsf{ev}(V) \\
&\text{E-EVT} && \\
& && P \fatsemi \mathcal{D}[\textbf{domTag}(\mathsf{ev}(W) \cdot \overrightarrow{e})\ \mathsf{t}\ U\ D] \longrightarrow P \parallel (\!(V_1\ W)\!) \parallel \cdots \parallel (\!(V_n\ W)\!) \fatsemi \mathcal{D}[\textbf{domTag}(\overrightarrow{e})\ \mathsf{t}\ U\ D] \\
& && \qquad \text{where } \mathsf{handlers}(\mathsf{ev}, U) = \overrightarrow{V} \\
&\text{E-STRUCT} && \mathcal{C} \longrightarrow \mathcal{C}' \quad \text{if } \mathcal{C} \equiv \mathcal{C}_1, \mathcal{C}_1 \longrightarrow \mathcal{C}_2, \text{ and } \mathcal{C}_2 \equiv \mathcal{C}' \\
&\text{E-LIFTP} && P \fatsemi D \longrightarrow P' \fatsemi D \quad \text{if } P \longrightarrow P'
\end{aligned}$$

Figure 4 Reduction rules for λ_{MVU} terms and configurations.

Processes. An *initialisation process* **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 **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* $(\!(M)\!)$ 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* $P \fatsemi D$, 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 $(A \times (A \to \mathsf{Html}(B)) \times ((B \times A) \to A))$. 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* ***run*** $M \fatsemi$ ***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 λ_{MVU} 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 diffing operation. Define $\mathsf{erase}(D)$ as the operation $\mathsf{erase}(\mathbf{domTag}(\overrightarrow{e})\ \mathsf{t}\ U\ D) = \mathbf{htmlTag}\ \mathsf{t}\ U\ (\mathsf{erase}(D))$, 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 $\mathsf{diff}(U, D) = D'$ if $\mathsf{erase}(D') = U$, 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 view and update functions and let $V_m = (\mathit{contents} = \texttt{""})$, $V_v = \mathsf{view}$, and $V_u = \mathsf{update}$. We extend the HTML syntactic sugar to pages, letting $[\![-]\!]$ be a desugaring function and $[\![\texttt{<t}\ \overrightarrow{a}\ \texttt{@}\ \overrightarrow{e}\texttt{>}\overrightarrow{D_{\mathsf{H}}}\texttt{</t>}]\!] = \mathbf{domTag}(\overrightarrow{e})\ \mathsf{t}\ [\![\overrightarrow{a}]\!]\ [\![\overrightarrow{D_{\mathsf{H}}}]\!]$.

We write $\mathcal{R}^+$ for the transitive closure of a relation $\mathcal{R}$. We begin by supplying the model, view, and update parameters to an initial configuration. By E-Run, we get an event loop process, and then term $V_v\ V_m$ reduces to the initial rendered HTML. By diffing against the empty page, we display the initial DOM page (E-Update).

$$
\begin{array}{l}
\quad \mathbf{run}\ (V_m, V_v, V_u) \fatsemi \mathbf{domEmpty} \\
\longrightarrow (\text{E-Run}) \quad \langle (V_m, V_v\ V_m) \mid (V_v, V_u) \rangle \fatsemi \mathbf{domEmpty} \\
\longrightarrow_{\mathsf{M}}^{+} \quad \langle (V_m, \\
\qquad \texttt{<input type = "text" value = ""} \\
\qquad\quad \mathsf{onInput} = \{\lambda \mathit{str}.\mathsf{UpdateBox}(\mathit{str})\}\texttt{></input>}\ ) \mid (V_v, V_u) \rangle \fatsemi \mathbf{domEmpty} \\
\qquad \texttt{<div></div>} \\
\longrightarrow (\text{E-Update}) \\
\quad \langle \mathbf{idle}\ V_m \mid (V_v, V_u) \rangle \fatsemi \quad \begin{array}{l} \texttt{<input type = "text" value = ""} \\ \quad \mathsf{onInput} = \{\lambda \mathit{str}.\mathsf{UpdateBox}(\mathit{str})\}\ @\ \epsilon\texttt{></input>} \\ \texttt{<div} \ @\ \epsilon\texttt{></div>} \end{array}
\end{array}
$$

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 *does* contain an onInput handler, which handles the input event by spawning $((\mathsf{UpdateBox}(\texttt{"k"})))$.

$$
\begin{array}{ll}
\longrightarrow^{+} (\text{E-Interact}) & \texttt{<input type = "text" value = ""} \\
\quad \langle \mathbf{idle}\ V_m \mid (V_v, V_u) \rangle \fatsemi & \quad \mathsf{onInput} = \{\lambda \mathit{str}.\mathsf{UpdateBox}(\mathit{str})\}\ @\ \mathsf{click}(()) \cdot \\
& \quad \mathsf{keyDown}(75) \cdot \mathsf{keyUp}(75) \cdot \mathsf{input}(\texttt{"k"})\texttt{></input>} \\
& \texttt{<div} \ @\ \epsilon\texttt{></div>} \\
\longrightarrow^{+} (\text{E-Evt}) & \texttt{<input type = "text" value = ""} \\
\quad \langle \mathbf{idle}\ V_m \mid (V_v, V_u) \rangle \fatsemi & \quad \mathsf{onInput} = \{\lambda \mathit{str}.\mathsf{UpdateBox}(\mathit{str})\}\ @\ \mathsf{input}(\texttt{"k"})\texttt{>} \\
& \texttt{</input>} \\
& \texttt{<div} \ @\ \epsilon\texttt{></div>} \\
\longrightarrow (\text{E-Evt}) & \qquad\qquad \texttt{<input type = "text" value = ""} \\
\quad \langle \mathbf{idle}\ V_m \mid (V_v, V_u) \rangle \parallel ((\mathsf{UpdateBox}(\texttt{"k"}))) \fatsemi & \qquad\quad \mathsf{onInput} = \{\lambda \mathit{str}.\mathsf{UpdateBox}(\mathit{str})\} \\
& \quad @\ \epsilon\texttt{></input>} \\
& \quad \texttt{<div} \ @\ \epsilon\texttt{></div>}
\end{array}
$$

Since $\mathsf{UpdateBox}(\texttt{"k"})$ 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 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:

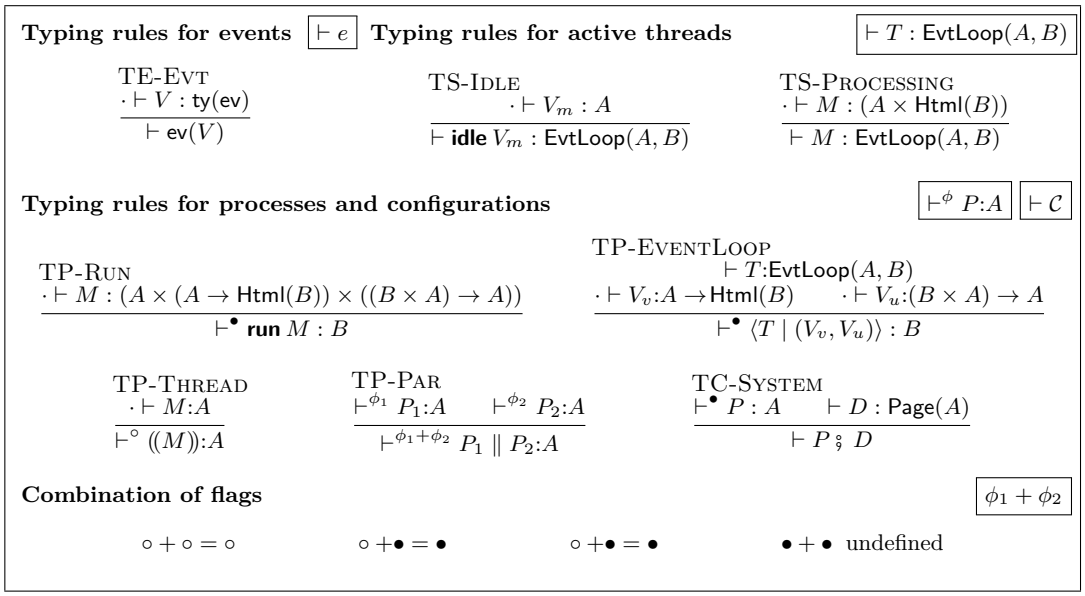


Figure 5 Runtime typing for λ_{MVU}.

$\longrightarrow$ (EP-Handle)

$\langle \mathsf{handle}(V_m, (V_v, V_u), \mathsf{UpdateBox}(\texttt{"k"})) \mid (V_v, V_u) \rangle \fatsemi$

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

$\longrightarrow^{+}_{\mathsf{M}}$

$(contents = \text{“}k\text{”})$,
$\langle$(
<input type = "text" value = "k"
onInput = $\{\lambda str.\mathsf{UpdateBox}(str)\}$>) $\mid (V_v, V_u) \rangle \fatsemi$
</input>
k

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

$\longrightarrow$ (E-Update)

$\langle \mathbf{idle}\ (contents = \texttt{"k"}) \mid (V_v, V_u) \rangle \fatsemi$

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

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 $\vdash T : \mathsf{EvtLoop}(A, B)$ can be read “Active thread T has model type A and message type B”. An idle thread $\mathbf{idle}\ V_m$ has type $\mathsf{EvtLoop}(A, B)$ if V_m has type A (TS-Idle). An active thread M currently processing a message has type $\mathsf{EvtLoop}(A, B)$ if M has type $(A \times \mathsf{Html}(B))$, i.e., computes a pair of a new model with type A and HTML which produces messages of type B (TS-Processing).

Judgement $\vdash^{\phi} P : A$ 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 $(\!(M)\!)$ has message type A if term M has type A (TP-Thread).

An initialisation process $\mathbf{run}\, M$ is well-typed if M is a product type where each component has the correct model, view, and update types. An event loop process $\langle T \mid (V_v, V_v) \rangle$ has message type B if its active thread T has model type A and message type B; its view function

V_v has type $A \to \mathsf{Html}(B)$; and its update function has type $(B \times A) \to A$ (TP-EventLoop). Thread flags ϕ ensure that there is precisely one initialisation process or event loop process in a process typeable under flag $\bullet$.

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

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.

▶ **Theorem 2** (Preservation). *If $\vdash \mathcal{C}$ and $\mathcal{C} \longrightarrow \mathcal{C}'$, then $\vdash \mathcal{C}'$.*

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.

▶ **Lemma 3** (Progress (Terms)). *If $\cdot \vdash M : A$, then either M is a value, or there exists some N such that $M \longrightarrow_M N$.*

Let $\longrightarrow_{\mathsf{E}}$ be the relation $\longrightarrow$ 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.

▶ **Theorem 4** (Event Progress). *If $\vdash \mathcal{C}$, either:*

1. *there exists some $\mathcal{C}'$ such that $\mathcal{C} \longrightarrow_E \mathcal{C}'$; or*
2. *$\mathcal{C} = \langle \mathbf{idle}\, V_m \mid (V_v, V_u) \rangle \fatsemi D$ where D cannot be written $\mathcal{D}[\mathbf{domTag}(\overrightarrow{e})\; t\; V\; W]$ for some non-empty $\overrightarrow{e}$.*

3 λ_{MVU} with Session Types

In this section, we extend λ_{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 $\mathsf{naïveFib}(x)$ be the naïve Fibonacci function and assuming an $\mathsf{intToString}$ function, we can write:

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

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

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

The type of the update function is changed to return a *pair* of an updated model and a command. In our case, the StartComputation message results in a pair of Nothing and **cmdSpawn** Result(naïveFib(1000)), which spawns Result(naïveFib(1000)) to evaluate in a separate thread. When the function (eventually) completes, the thread will have evaluated to a Result message, which can be processed by the update function to update the model and display the result.

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, λ_{MVU} as defined so far does not support linearity. Consider handle:

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

The updated model, 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:

$$\mathsf{extract} : A \to (A \times B)$$

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):

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

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

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

```
Model ≜ (Bool × Chan(Ping) × Chan(Pong))

view : Model → Html(Message)
view ≜ λ(pinging, _, _).
  let attr =
    if pinging then
      attrEmpty
    else
      attr "disabled" "true" in
  html
    <html>
      <body>
        <button {attr} onClick = {λ().Click}>
          Send Ping!
        </button>
      </body>
    </html>
```

```
Message ≜ Click | Ponged

update : (Message × Model) → Model
update ≜ λ(msg, (_, pingCh, pongCh)).
  case msg {
    Click ↦
      let cmd =
        cmdSpawn ( send (Ping, pingCh);
                   let Pong = receive pongCh in
                   Ponged) in
      ((false, pingCh, pongCh), cmd)
    Ponged ↦ ((true, pingCh, pongCh), cmdEmpty)
  }

server : (Chan(Ping) × Chan(Pong)) → (1 → A)
server ≜ λ(pingCh, pongCh).
  (rec f() .
    let Ping = receive pingCh in
    send (Pong, pongCh); f ())
```

Figure 6 PingPong application using simply-typed channels.

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 λ_{MVU} application. Our web client consists of a button which, when clicked, triggers the sending of a Ping message to the server. Once clicked, the button is disabled. The server then receives the Ping message and responds with a 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 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 Ping messages; and a channel for Pong messages. There are two types of UI message: Click denotes that the button has been clicked, and Ponged denotes that a Pong message has been received along the Pong channel.

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

Even in this simple example, it is very easy to communicate incorrectly: if the client neglected to send a Ping message before trying to receiving a Pong message along *pongCh*, then the command would hang forever and the GUI would never re-enable the button. A similar situation would arise if the server received the Ping message but failed to respond.

```
PingPong ≜ μt.!Ping.?Pong.t
UModel ≜ UPinging | UWaiting
Model ≜ Pinging(PingPong) | Waiting
Message ≜ Click | Ponged(PingPong)

view : UModel → Html(Message)
view ≜ λuModel.
  let attr =
    case uModel {
      UPinging ↦ attrEmpty
      UWaiting ↦ attr "disabled" "true"
    } in
  html
    <html>
      <body>
        <button {attr} onClick = {λ().Click}>
          Send Ping!
        </button>
      </body>
    </html>

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

update : (Message × Model) →
    (Model × Cmd(Message))
update ≜ λ(msg, model).
  case msg {
    Click ↦
      handleClick(model)
    Ponged(c) ↦
      handlePonged(model, c)
  }

extract : Model → (Model × UModel)
extract ≜ λmodel.
  case model {
    Pinging(c) ↦ (Pinging(c), UPinging)
    Waiting ↦ (Waiting, UWaiting)
  }

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

Figure 7 PingPong application.

Session types. Session types S range over output $!A.S$, input $?A.S$, the completed session End, recursive session types $\mu t.S$, 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 [28, 13]. The **send** constant sends a value of type A over an endpoint of type $!A.S$ and returns the continuation of the session, S. The **close** constant closes a completed session endpoint. The **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 **cancel** constant allows an endpoint to be discarded safely [35, 20].

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

$$\textbf{send} : (A \times !A.S) \to S \qquad \textbf{receive} : ?A.S \to (A \times S) \qquad \textbf{close} : \mathsf{End} \to \mathbf{1} \qquad \textbf{cancel} : S \to \mathbf{1}$$

Figure 7 shows the PingPong client written in λ_{MVU}. We can encode the PingPong protocol as a session type, $\mu t.!\mathsf{Ping}.?\mathsf{Pong}.t$. The Model type encodes the two states of the application: 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 Waiting is the state where the button is disabled and awaiting a Pong message from the other party. The UModel type is the unrestricted model type which does not include the session channel. Again, the Message type encodes the UI messages 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.

Pinging state

```
PModel ≜ Pinging(PingPong)
PUModel ≜ 1
PMessage ≜ Click
pView : PUModel → Html(PMessage)
pView ≜ λ(). html
  <html>
    <body>
      <button onClick = {λ().Click}>
        Send Ping!
      </button>
    </body>
  </html>
pUpdate : (PMessage × PModel) →
    Transition(PModel, PMessage)
pUpdate ≜ λ(Click, Pinging(c)).
  let cmd =
    cmdSpawn ( let c = send (Ping, c) in
               let (pong, c) = receive c in
               Ponged(c)) in
  transition Waiting wView wUpdate wExtract cmd
pExtract : PModel → (PModel × PUModel)
pExtract ≜ λx.(x, ())
```

Waiting state

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

wView : WUModel → Html(WMessage)
wView ≜ λ(). html
  <html>
    <body>
      <button disabled = "true">
        Send Ping!
      </button>
    </body>
  </html>

wUpdate : (WMessage × WModel) →
    Transition(WModel, WMessage)
wUpdate ≜ λ(Ponged(c), Waiting).
  transition Pinging(c) pView
    pUpdate pExtract cmdEmpty

wExtract : WModel →
    (WModel × WUModel)
wExtract ≜ λx.(x, ())
```

Figure 8 PingPong application using model transitions.

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.

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.

$$
\begin{array}{lrcl}
\text{Kinds} & \kappa & ::= & \mathsf{L} \mid \mathsf{U} \\
\text{Types} & A, B, C & ::= & \mathbf{1} \mid A \to^{\kappa} B \mid A \times B \mid A + B \mid \mathsf{String} \mid \mathsf{Int} \mid S \\
& & \mid & \mathsf{Html}(A) \mid \mathsf{Attr}(A) \mid \mathsf{Cmd}(A) \mid \mathsf{Transition}(A, B) \\
\text{Session types} & S & ::= & !A.S \mid ?A.S \mid \mu t.S \mid t \mid \bar{t} \mid \mathsf{End} \\
\\
\text{Terms} & L, M, N & ::= & x \mid \lambda x.M \mid M\ N \mid K\ M \mid () \mid s \mid n \\
& & \mid & (M, N) \mid \textbf{let}\ (x, y) = M\ \textbf{in}\ N \\
& & \mid & \textbf{inl}\ x \mid \textbf{inr}\ x \mid \textbf{case}\ L\ \{\textbf{inl}\ x \mapsto M; \textbf{inr}\ y \mapsto N\} \\
& & \mid & \textbf{htmlTag}\ \mathsf{t}\ M\ N \mid \textbf{htmlText}\ M \\
& & \mid & \textbf{attr}\ ak\ M \mid \textbf{attrEmpty} \\
& & \mid & \textbf{cmdSpawn}\ M \mid \textbf{cmdEmpty} \mid M \star N \\
& & \mid & \textbf{transition}\ M_m\ M_v\ M_u\ M_e\ M_c \mid \textbf{noTransition}\ M_m\ M_c \\
& & \mid & \textbf{raise} \mid \textbf{try}\ L\ \textbf{as}\ x\ \textbf{in}\ M\ \textbf{otherwise}\ N \\
\text{Constants} & K & ::= & \textbf{send} \mid \textbf{receive} \mid \textbf{new} \mid \textbf{cancel} \mid \textbf{close}
\end{array}
$$

Figure 9 Syntax of extended calculus.

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 λ_{MVU} 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 λ_{MVU} extended with commands, linearity, and transitions.

Types and kinds. To support linearity, types are assigned *kinds*, ranged over by κ. 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 Cmd(A) which is the type of a command which produces messages of type A. To support transitions, we introduce type 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 **cmdSpawn** M is a command which can spawn term M as a thread, and is monoidally composable using $\star$ and **cmdEmpty**.

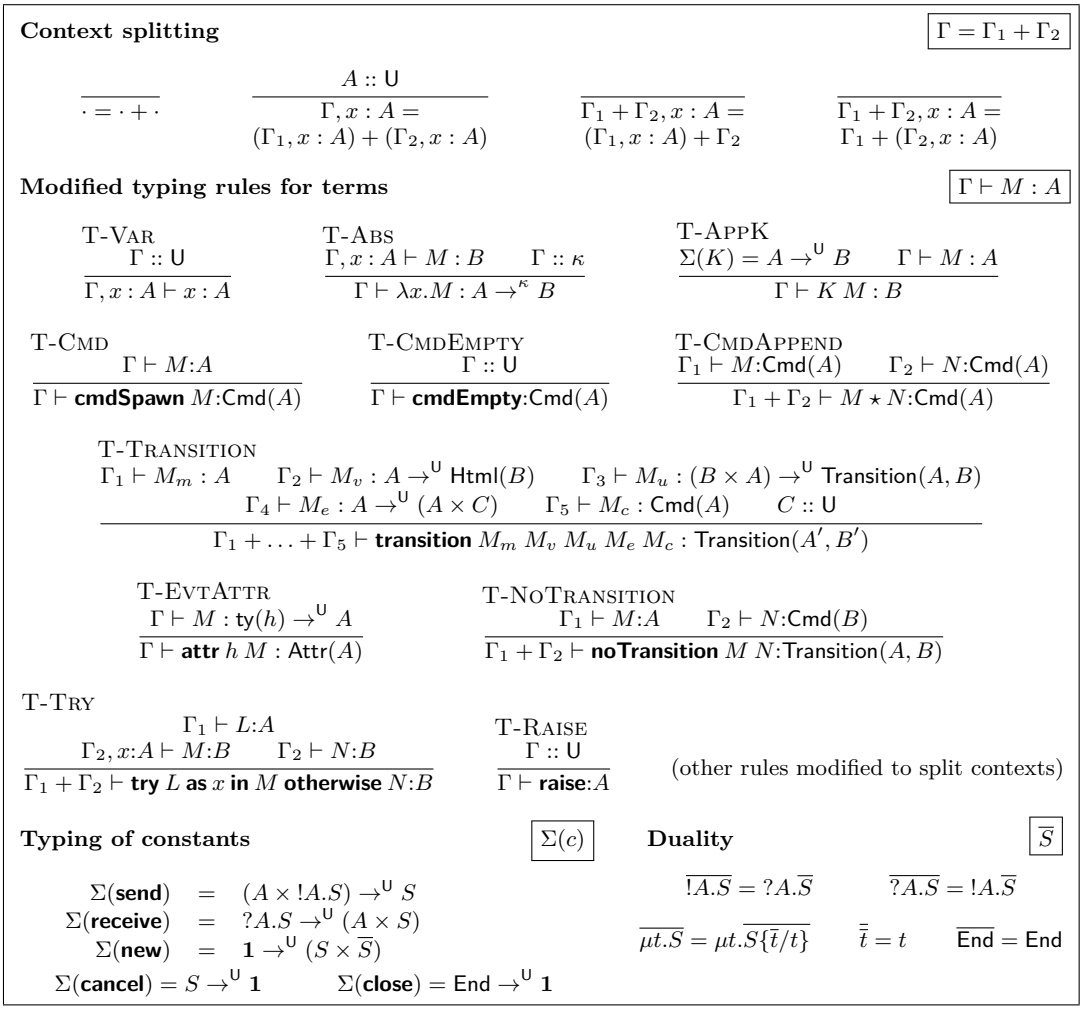


Figure 10 Term typing for extended calculus.

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** $M_m\ M_v\ M_u\ M_e\ M_c$ 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, as described in §3.2. We also introduce an application form for constants, $K\ M$.

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 **try** L **as** x **in** M **otherwise** 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 κ 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 $\mathsf{U} \leq \mathsf{L}$ 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 $x : A \in \Gamma$.

▶ **Definition 5** (Kinding and subkinding). *We define the* subkinding *relation as the reflexive relation on kinds $\leq$ such that $\mathsf{U} \leq \mathsf{L}$. We define the* kinding *relation $A :: \kappa$ as the largest relation between types and kinds such that:*

- $A :: \kappa'$ *if* $A :: \kappa$ *and* $\kappa \leq \kappa'$
- $S :: \mathsf{L}$
- $A :: \mathsf{U}$ *if* $A \in \{\mathbf{1}, \mathsf{String}, \mathsf{Int}, \mathsf{Html}(B), \mathsf{Attr}(B)\}$
- $A \to^{\kappa} B :: \kappa$
- $\mathsf{Cmd}(A) :: \kappa$ *if* $A :: \kappa$
- $C :: \kappa$ *if* $C \in \{A \times B, A + B, \mathsf{Transition}(A, B)\}$ *and both* $A :: \kappa$ *and* $B :: \kappa$

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

Rule T-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 $\mathsf{Transition}(A', B')$ need not match the types of the new model and functions. Rule T-NoTransition assigns term **noTransition** $M\ N$ type $\mathsf{Transition}(A, B)$ if new model M has type A, and N is a command of type $\mathsf{Cmd}(B)$. 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.

Runtime syntax

Runtime names	c, d		
Values	U, V, W	$::=$	$\cdots \mid c \mid \textbf{cmdSpawn}\ M \mid \textbf{noTransition}\ V\ W$
		$\mid$	$\textbf{transition}\ V_m\ V_v\ V_u\ V_e\ V_c$
Active thread	T	$::=$	$\textbf{idle}\ V_m \mid \textbf{updating}\ M \mid \textbf{extracting}[V_c]\ M$
		$\mid$	$\textbf{extractingT}[F, V_c]\ M \mid \textbf{rendering}[V_m, V_c]\ M$
		$\mid$	$\textbf{transitioning}[V_m, F, V_c]\ M$
Versions	ι		
Processes	P, Q	$::=$	$\textbf{run}\ M \mid \langle T \mid F \rangle_\iota \mid (\!(M)\!)_\iota \mid P \parallel Q$
		$\mid$	$(\nu cd)P \mid \lfloor M \rfloor \mid \lightning c \mid \textbf{halt}$
Function state	F	$::=$	(V_v, V_u, V_e)
Configurations	$\mathcal{C}$	$::=$	$P \fatsemi D$
Process contexts	$\mathcal{P}$	$::=$	$[\,] \mid \mathcal{P} \parallel P \mid (\nu cd)\mathcal{P}$
Evaluation contexts	E	$::=$	$\cdots \mid K\ E \mid \textbf{noTransition}\ E\ M \mid \textbf{noTransition}\ V\ E$
		$\mid$	$\textbf{transition}\ E\ M_v\ M_u\ M_e\ M_c \mid \cdots \mid \textbf{transition}\ V_m\ V_v\ V_u\ V_e\ E$
		$\mid$	$\textbf{try}\ E\ \textbf{as}\ x\ \textbf{in}\ M\ \textbf{otherwise}\ N$
Pure contexts	E_P	$::=$	(as E, but without exception handling frames)
Active thread contexts	$\mathcal{T}_\mathsf{A}$	$::=$	$\textbf{updating}\ E \mid \textbf{rendering}[V_m, V_c]\ E \mid \textbf{extracting}[V_c]\ E$
		$\mid$	$\textbf{extractingT}[V_c, F']\ E \mid \textbf{transitioning}[V_m, F', V_c]\ E$
Pure active thread contexts	$\mathcal{T}_\mathsf{P}$	$::=$	(as $\mathcal{T}_\mathsf{A}$, but for pure contexts)
Thread contexts	$\mathcal{T}$	$::=$	$\textbf{run}\ E \mid \langle \mathcal{T}_\mathsf{A} \mid F \rangle_\iota \mid (\!(E)\!)_\iota \mid \lfloor E \rfloor$

Active thread state machine

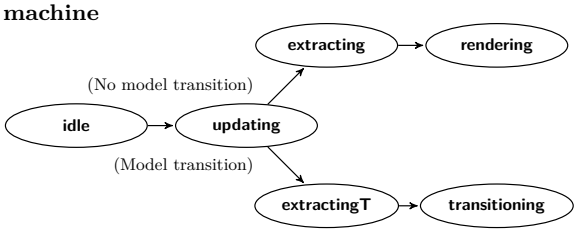


Figure 11 Runtime syntax for extended calculus.

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 $\textbf{rendering}[V_m, V_c]\ M$ state records the new model V_m and the command to be executed upon updating the page V_c.

We introduce four new types of process. To model client-server communication, we introduce server processes $\lfloor M \rfloor$ 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:

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

Additional term reduction rule $\boxed{M \longrightarrow_{\mathsf{M}} N}$

E-Try
$\mathbf{try}\ V\ \mathbf{as}\ x\ \mathbf{in}\ M\ \mathbf{otherwise}\ N \longrightarrow_{\mathsf{M}} M\{V/x\}$

Additional meta-level definitions

$$\begin{aligned}\mathsf{procs}(\mathbf{cmdEmpty}) &= \epsilon\\ \mathsf{procs}(\mathbf{cmdSpawn}\ M) &= M\\ \mathsf{procs}(V \star W) &= \mathsf{procs}(V) \cdot \mathsf{procs}(W)\end{aligned}$$

Equivalence of processes $\boxed{P \equiv P'}$

$$(\nu cd)(\nu c'd')P \equiv (\nu c'd')(\nu cd)P \qquad P \parallel ((\nu cd)Q) \equiv (\nu cd)(P \parallel Q) \quad \text{if } c,d \notin \mathsf{fn}(P) \qquad (\nu cd)P \equiv (\nu dc)P$$

$$P_1 \parallel P_2 \equiv P_2 \parallel P_1 \qquad P_1 \parallel (P_2 \parallel P_3) \equiv (P_1 \parallel P_2) \parallel P_3 \qquad (\nu cd)(\lightning c \parallel \lightning d) \parallel P \equiv P \qquad \lfloor () \rfloor \parallel P \equiv P$$

Reduction of processes $\boxed{P \longrightarrow P'}$

MVU reduction rules

$$\begin{aligned}
&\text{E-Discard} && \langle T \mid F\rangle_\iota \parallel ((V))_{\iota'} \longrightarrow \langle T \mid F\rangle_\iota \parallel \lightning V \quad \text{if } \iota \neq \iota'\\
&\text{E-DiscardHalt} && \mathbf{halt} \parallel ((V))_\iota \longrightarrow \mathbf{halt} \parallel \lightning V\\
&\text{E-Handle} && \langle \mathbf{idle}\ V_m \mid (V_v, V_u, V_e)\rangle_\iota \parallel ((V))_\iota \longrightarrow \langle \mathbf{updating}\ V_u\ (V, V_m) \mid (V_v, V_u, V_e)\rangle_\iota\\
&\text{E-Extract} && \langle \mathbf{updating}\ (\mathbf{noTransition}\ V_m\ V_c) \mid F\rangle_\iota \longrightarrow \langle \mathbf{extracting}[V_c]\ (V_e\ V_m) \mid F\rangle_\iota\\
& && \quad \text{where } F = (V_v, V_u, V_e)\\
&\text{E-ExtractT} && \langle \mathbf{updating}\ (\mathbf{transition}\ V_m\ V_v\ V_u\ V_e\ V_c) \mid F\rangle_\iota \longrightarrow \langle \mathbf{extracting}\mathbf{T}[(V_v, V_u, V_e), V_c]\ (V_e\ V_m) \mid F\rangle_\iota\\
&\text{E-Render} && \langle \mathbf{extracting}[V_c]\ (V_m, V_{um}) \mid F\rangle_\iota \longrightarrow \langle \mathbf{rendering}[V_m, V_c]\ (V_v\ V_{um}) \mid F\rangle_\iota\\
& && \quad \text{where } F = (V_v, V_u, V_e)\\
&\text{E-RenderT} && \langle \mathbf{extracting}\mathbf{T}[F', V_c]\ (V_m, V_{um}) \mid F\rangle_\iota \longrightarrow \langle \mathbf{transitioning}[V_m, F', V_c]\ (V_v\ V_{um}) \mid F\rangle_\iota\\
& && \quad \text{where } F' = (V_v, V_u, V_e)
\end{aligned}$$

Session reduction rules

$$\begin{aligned}
&\text{E-New} && \mathcal{T}[\mathbf{new}()] \longrightarrow (\nu cd)(\mathcal{T}[(c,d)]) \quad \text{where } c,d \text{ fresh}\\
&\text{E-Comm} && (\nu cd)(\mathcal{T}[\mathbf{send}\ (V,c)] \parallel \mathcal{T}'[\mathbf{receive}\ d]) \longrightarrow (\nu cd)(\mathcal{T}[c] \parallel \mathcal{T}'[(V,d)])\\
&\text{E-Close} && (\nu cd)(\mathcal{T}[\mathbf{close}\ c] \parallel \mathcal{T}'[\mathbf{close}\ d]) \longrightarrow \mathcal{T}[()] \parallel \mathcal{T}'[()]\\
&\text{E-Cancel} && \mathcal{T}[\mathbf{cancel}\ c] \longrightarrow \mathcal{T}[()] \parallel \lightning c\\
&\text{E-SendZap} && (\nu cd)(\mathcal{T}[\mathbf{send}\ (V,c)] \parallel \lightning d) \longrightarrow (\nu cd)(\mathcal{T}[\mathbf{raise}] \parallel \lightning c \parallel \lightning V \parallel \lightning d)\\
&\text{E-RecvZap} && (\nu cd)(\mathcal{T}[\mathbf{receive}\ c] \parallel \lightning d) \longrightarrow (\nu cd)(\mathcal{T}[\mathbf{raise}] \parallel \lightning c \parallel \lightning d)\\
&\text{E-CloseZap} && (\nu cd)(\mathcal{T}[\mathbf{close}\ c] \parallel \lightning d) \longrightarrow (\nu cd)(\mathcal{T}[\mathbf{raise}] \parallel \lightning c \parallel \lightning d)
\end{aligned}$$

Exception reduction rules

$$\begin{aligned}
&\text{E-RaiseH} && \mathcal{T}[\mathbf{try}\ E_{\mathsf{P}}[\mathbf{raise}]\ \mathbf{as}\ x\ \mathbf{in}\ M\ \mathbf{otherwise}\ N] \longrightarrow \mathcal{T}[N] \parallel \lightning E_{\mathsf{P}}\\
&\text{E-RaiseURun} && \mathbf{run}\ (E_{\mathsf{P}}[\mathbf{raise}]) \longrightarrow \mathbf{halt} \parallel \lightning E_{\mathsf{P}}\\
&\text{E-RaiseUMain} && \langle \mathcal{T}_{\mathsf{P}}[\mathbf{raise}] \mid F\rangle_\iota \longrightarrow \mathbf{halt} \parallel \lightning \mathcal{T}_{\mathsf{P}}\\
&\text{E-RaiseUThread} && ((E_{\mathsf{P}}[\mathbf{raise}]))_\iota \longrightarrow \lightning E_{\mathsf{P}}\\
&\text{E-RaiseUServer} && \lfloor E_{\mathsf{P}}[\mathbf{raise}] \rfloor \longrightarrow \lightning E_{\mathsf{P}}
\end{aligned}$$

Administrative reduction rules

$$\begin{aligned}
&\text{E-LiftT} && \mathcal{T}[M] \longrightarrow \mathcal{T}[N] && \text{if } M \longrightarrow_{\mathsf{M}} N\\
&\text{E-Nu} && (\nu ab)P \longrightarrow (\nu ab)P' && \text{if } P \longrightarrow P'\\
&\text{E-Par} && P_1 \parallel P_2 \longrightarrow P_1' \parallel P_2 && \text{if } P_1 \longrightarrow P_1'
\end{aligned}$$

Figure 12 Reduction rules for extended calculus (1).

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* ι 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 $((M))_{\iota'}$ where $\iota \neq \iota'$ can be of arbitrary type as it will be discarded. We write $\mathsf{version}(P) = \iota$ 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 the procs meta-definition returns a sequence of processes to be spawned by a command. Process equivalence is extended to allow commutativity of name restrictions, reordering of names in a binder, and scope extrusion. The final "garbage collection" equivalences $(\nu cd)(\lightning c \parallel \lightning d) \parallel P \equiv P$ and $\lfloor () \rfloor \parallel P \equiv P$ allow us to discard a channel where both endpoints have been cancelled, and a completed server thread, respectively.

Figure 12 details the extended MVU process reduction rules.

Reduction of configurations $\boxed{\mathcal{C} \longrightarrow \mathcal{C}'}$

E-Run
$$\mathcal{P}[\mathbf{run}\ (V_m, V_v, V_u, V_e, V_c, \lambda().M)] \fatsemi D \longrightarrow \langle \mathbf{extracting}[V_c]\ (V_e\ V_m) \mid (V_v, V_u, V_e)\rangle_0 \parallel \lfloor M \rfloor \fatsemi D$$

E-Update
$$\mathcal{P}[\langle \mathbf{rendering}[V_m', V_c]\ U \mid F\rangle_\iota] \fatsemi D \longrightarrow \mathcal{P}[\langle \mathbf{idle}\ V_m' \mid F\rangle_\iota \parallel (\!(M_1)\!)_\iota \parallel \cdots \parallel (\!(M_n)\!)_\iota] \fatsemi D'$$
where $\mathsf{diff}(U, D) = D'$ and $\mathsf{procs}(V_c) = \overrightarrow{M}$

E-Transition
$$\mathcal{P}[\langle \mathbf{transitioning}[V_m, F', V_c]\ U \mid F\rangle_\iota] \fatsemi D \longrightarrow \mathcal{P}[\langle \mathbf{idle}\ V_m \mid F'\rangle_{\iota'} \parallel (\!(M_1)\!)_{\iota'} \parallel \cdots \parallel (\!(M_n)\!)_{\iota'}] \fatsemi D'$$
where $\iota' = \iota + 1, \mathsf{diff}(U, D) = D'$ and $\mathsf{procs}(V_c) = \overrightarrow{M}$

E-Evt
$$P \fatsemi \mathcal{D}[\mathbf{domTag}(\mathsf{ev}(W) \cdot \overrightarrow{e})\ \mathsf{t}\ U\ D] \longrightarrow P \parallel (\!(V_1\ W)\!)_\iota \parallel \cdots \parallel (\!(V_n\ W)\!)_\iota \fatsemi \mathcal{D}[\mathbf{domTag}(\overrightarrow{e})\ \mathsf{t}\ U\ D]$$
where $\mathsf{handlers}(\mathsf{ev}, U) = \overrightarrow{V}$ and $\mathsf{version}(P) = \iota$

(E-Interact, E-Struct, E-LiftP unchanged) **Cancellation of pure active thread contexts** $\boxed{\lightning \mathcal{T}_\mathsf{P}}$

$$\lightning \mathbf{updating}\ E_\mathsf{P} = \lightning E_\mathsf{P} \qquad \lightning \mathbf{rendering}[V_m, V_c]\ E_\mathsf{P} = \lightning V_m \parallel \lightning V_c \parallel \lightning E_\mathsf{P} \qquad \lightning \mathbf{extracting}[V_c]\ E_\mathsf{P} = \lightning V_c \parallel \lightning E_\mathsf{P}$$

$$\lightning \mathbf{extractingT}[V_c, F]\ E_\mathsf{P} = \lightning V_c \parallel \lightning E_\mathsf{P} \qquad \lightning \mathbf{transitioning}[V_m, F, V_c]\ E_\mathsf{P} = \lightning V_m \parallel \lightning V_c \parallel \lightning E_\mathsf{P}$$

Figure 13 Reduction rules for extended calculus (2).

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-ExtractT, 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 reduction. 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 **halt** configuration and cancelling any linear resources in the aborted context. Rules E-RaiseUThread and E-RaiseUServer 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.

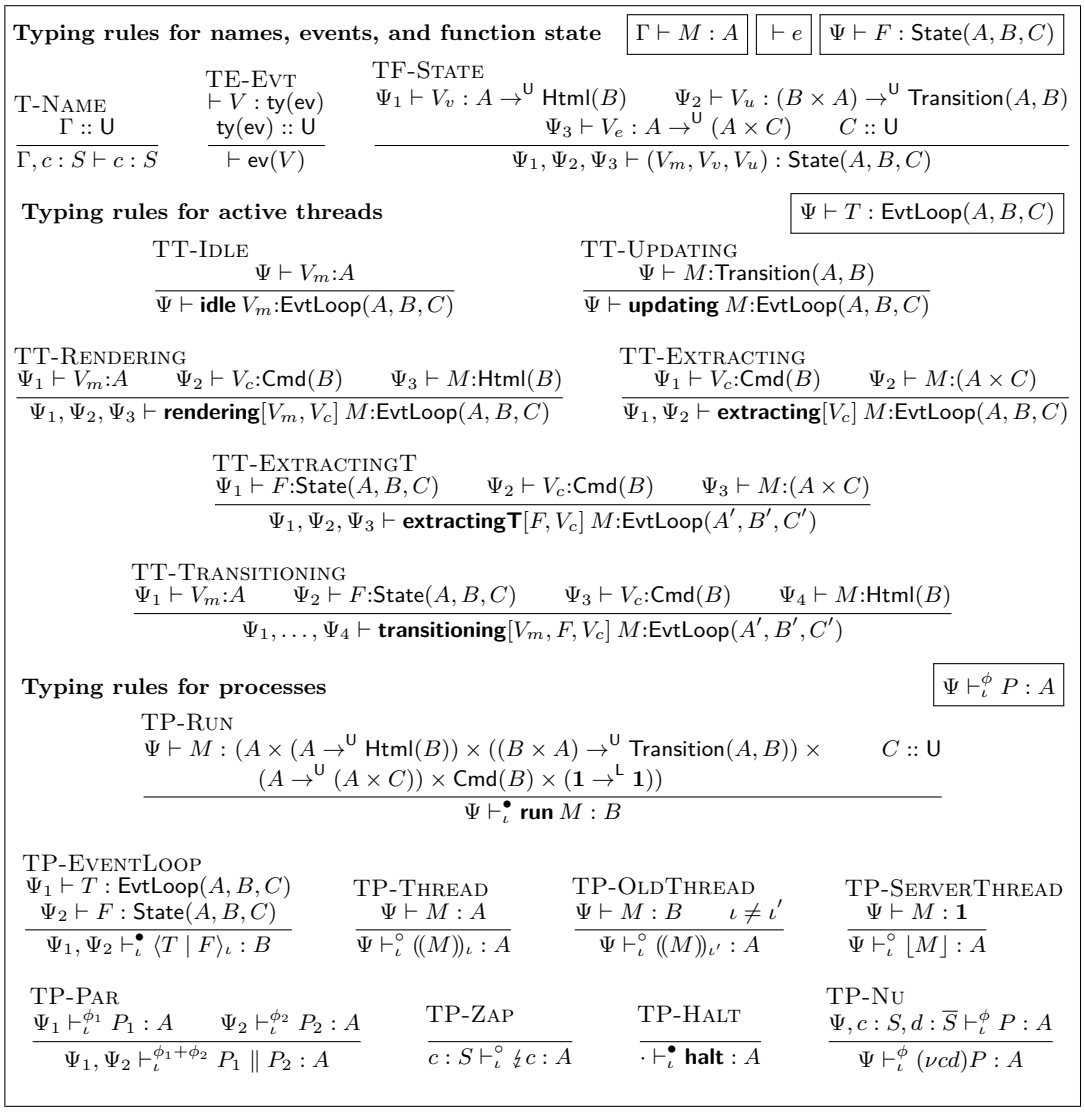


Figure 14 Runtime typing for extended calculus.

3.4.3 Metatheory

Runtime typing. Figure 14 shows the runtime typing rules for the extended calculus. Rule T-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 $\mathsf{Transition}(A, B)$), and that any recorded values have the correct types.

Let Ψ range over environments containing only runtime names: $\Psi ::= \cdot \mid \Psi, c : S$. We write Ψ_1, Ψ_2 for the disjoint union of environments Ψ_1 and Ψ_2.

We modify the shape of the process typing judgement to $\Psi \vdash_\iota^\phi P : A$, which can be read "under typing environment Ψ and thread flag ϕ, process P has type A and version ι". 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 $(\!(M)\!)_\iota$ 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 $(\nu cd)P$ by adding c and d with dual session types into the typing environment.

Properties. The extended calculus satisfies preservation.

▶ **Theorem 7** (Preservation). *If $\vdash \mathcal{C}$ and $\mathcal{C} \longrightarrow \mathcal{C}'$, then $\vdash \mathcal{C}'$.*

Although session types rule out deadlock within a single session, without imposing a tree-like structure on processes [45, 31] (which is too inflexible for our purposes) or using techniques such as channel priorities [37, 39, 29], 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. $(\nu cd)(\mathcal{T}[\textbf{\textit{send}}\,(V, c)] \parallel \mathcal{T}'[\textbf{\textit{send}}\,(W, d)])$
2. $(\nu cd)(\mathcal{T}[\textbf{\textit{send}}\,(V, c)] \parallel \mathcal{T}'[\textbf{\textit{close}}\,d])$
3. $(\nu cd)(\mathcal{T}[\textbf{\textit{receive}}\,c] \parallel \mathcal{T}'[\textbf{\textit{receive}}\,d])$
4. $(\nu cd)(\mathcal{T}[\textbf{\textit{receive}}\,c] \parallel \mathcal{T}'[\textbf{\textit{close}}\,d])$

Configuration typing ensures error-freedom.

▶ **Theorem 9** (Error-freedom). *If $\Psi \vdash^{\phi}_{\iota} P : A$, 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 λ_{MVU}. We begin by classifying the notion of a *blocked* thread, which is a thread blocked on a communication action.

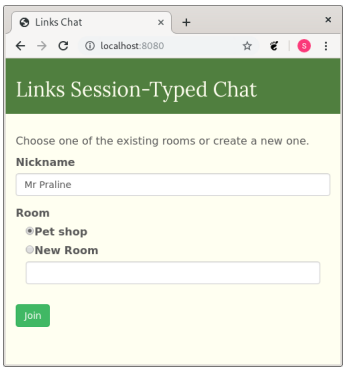
▶ **Definition 10** (Blocked thread). *We say a thread $\mathcal{T}[M]$ is* blocked *if either $M = \textbf{\textit{send}}(V, W)$, $M = \textbf{\textit{receive}}\,V$, or $M = \textbf{\textit{close}}\,V$.*

Let us refer to **halt**, $\langle T \mid F \rangle_{\iota}$, and **run** M as *main threads*, and $(\!(M)\!)_{\iota}$, $\lfloor M \rfloor$, and $\lightning c$ as *auxiliary threads*. Each well-typed configuration has precisely one main thread.

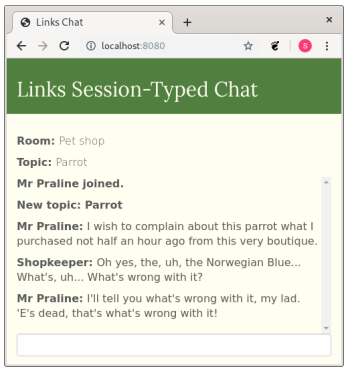
We can now classify the notion of progress enjoyed by the extended calculus. Either the configuration can reduce; is waiting for an event; has halted due to an unhandled exception; or is deadlocked. Again, let $\longrightarrow_{\mathsf{E}}$ be the $\longrightarrow$ relation without E-Interact.

▶ **Theorem 11** (Weak Event Progress). *Suppose $\vdash \mathcal{C}$. Either there exists some $\mathcal{C}'$ such that $\mathcal{C} \longrightarrow \mathcal{C}'$, or there exists some $\mathcal{C}'$ such that $\mathcal{C} \equiv \mathcal{C}'$ and:*

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



(a) Login



(b) Chat

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 |&];

typename 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 |&];

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

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

Figure 15 Chat server application.

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.

```
typename SelectedRoom =
  [| NewRoom | SelectedRoom: String |];
typename NotConnectedModel =
  (nickname: String, rooms: [RoomName],
   selectedRoom: SelectedRoom,
   newRoomText: RoomName, error: Maybe(Error));
```

```
typename NCModel =
  (ClientSelect, NotConnectedModel);
typename NCMessage = [|
 [| UpdateNickname: Nickname
  | UpdateSelectedRoom: SelectedRoom
  | UpdateNewRoom: RoomName | SubmitJoinRoom |];
```

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 $(\mathsf{Model} \times \mathsf{Event}) \to \mathsf{Model}$. 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 λ_{MVU} (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 λ_{MVU} (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, λ_{MVU}. 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

1 Elm: A delightful language for reliable webapps, 2019. Accessed on 2019-07-04. URL: `http://www.elm-lang.org`.

2 React – a JavaScript library for building user interfaces, 2019. Accessed on 2019-09-02. URL: `http://www.reactjs.org`.

3 WebSharper.Mvu, 2019. Accessed on 2019-07-04. URL: `https://github.com/dotnet-websharper/mvu`.

4 Flux, 2020. Accessed on 2020-01-08. URL: `https://facebook.github.io/flux/`.

5 Redux - a predictable state container for JavaScript apps, 2020. Accessed on 2020-01-08. URL: `https://redux.js.org/`.

6 Robert Atkey. Parameterised notions of computation. *J. Funct. Program.*, 19(3-4):335–376, 2009.

7 Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, volume 6269 of *Lecture Notes in Computer Science*, pages 222–236. Springer, 2010.

8 Iliano Cervesato and Frank Pfenning. A linear logical framework. In *LICS*, pages 264–275. IEEE Computer Society, 1996.

9 Adam Chlipala. Ur/Web: A simple model for programming the web. In *POPL*, pages 153–165. ACM, 2015.

10 Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *FMCO*, volume 4709 of *Lecture Notes in Computer Science*, pages 266–296. Springer, 2006.

11 Evan Czaplicki. Farewell to FRP, 2016. Accessed on 2019-09-02. URL: `https://elm-lang.org/news/farewell-to-frp`.

12 Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for GUIs. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 411–422. ACM, 2013. `doi:10.1145/2491956.2462161`.

13 Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. *Inf. Comput.*, 256:253–286, 2017.

14 Conal Elliott and Paul Hudak. Functional reactive animation. In *ICFP*, pages 263–273. ACM, 1997.

15 Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. A functional I/O system or, fun for freshman kids. In *ICFP*, pages 47–58. ACM, 2009.

16 Ian Fette and Alexey Melnikov. The WebSocket protocol, 2011.

17 Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: a programming environment for Scheme. *J. Funct. Program.*, 12(2):159–182, 2002.

18 Simon Fowler. Model-View-Update-Communicate: Session types meet the Elm architecture (Extended version), 2019. `arXiv:1910.11108`.

19 Simon Fowler, Loïc Denuzière, and Adam Granicz. Reactive single-page applications with dynamic dataflow. In *PADL*, volume 9131 of *Lecture Notes in Computer Science*, pages 58–73. Springer, 2015.

20 Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional asynchronous session types: session types without tiers. *PACMPL*, 3(POPL):28:1–28:29, 2019. `doi:10.1145/3290341`.

21 Simon J. Gay and Vasco Thudichum Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010.

22 Ido Green. *Web Workers - Multithreaded Programs in JavaScript*. O'Reilly, 2012.

23 Kohei Honda. Types for dyadic interaction. In *CONCUR*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993.

24 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998.

25 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016.

26 Anupam Jain. Concur, 2019. Accessed on 2019-09-02. URL: `https://ajnsit.github.io/concur`.

27 Jonathan King, Nicholas Ng, and Nobuko Yoshida. Multiparty session type-safe web development with static linearity. In *PLACES@ETAPS*, volume 291 of *EPTCS*, pages 35–46, 2019.

28 Naoki Kobayashi. Type systems for concurrent programs. In *10th Anniversary Colloquium of UNU/IIST*, volume 2757 of *Lecture Notes in Computer Science*, pages 439–453. Springer, 2002.

29 Naoki Kobayashi. A new type system for deadlock-free processes. In *CONCUR*, volume 4137 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2006.

30 Glenn E. Krasner and Stephen T. Pope. A Cookbook for using the Model-view Controller user interface paradigm in Smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, August 1988. URL: `http://dl.acm.org/citation.cfm?id=50757.50759`.

31 Sam Lindley and J. Garrett Morris. A semantics for propositions as sessions. In *ESOP*, volume 9032 of *Lecture Notes in Computer Science*, pages 560–584. Springer, 2015.

32 Sam Lindley and J. Garrett Morris. Talking bananas: structural recursion for session types. In *ICFP*, pages 434–447. ACM, 2016.

33 Sam Lindley and J Garrett Morris. Lightweight functional session types. *Behavioural Types: from Theory to Tools. River Publishers*, pages 265–286, 2017.

34 Leo A. Meyerovich, Arjun Guha, Jacob P. Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: a programming language for Ajax applications. In *OOPSLA*, pages 1–20. ACM, 2009.

35 Dimitris Mostrous and Vasco T. Vasconcelos. Affine sessions. *Logical Methods in Computer Science*, 14(4), 2018. `doi:10.23638/LMCS-14(4:14)2018`.

36 Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. *Theor. Comput. Sci.*, 364(3):338–356, 2006.

37 Luca Padovani. Deadlock and lock freedom in the linear π-calculus. In *CSL-LICS*, pages 72:1–72:10. ACM, 2014.

38 Luca Padovani. Context-free session type inference. In *ESOP*, volume 10201 of *Lecture Notes in Computer Science*, pages 804–830. Springer, 2017.

39 Luca Padovani and Luca Novara. Types for deadlock-free higher-order programs. In *FORTE*, volume 9039 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2015.

40 Adam Pedley. Functional Model-View-Update Architecture for Flutter, 2019. Accessed on 2019-09-24. URL: `https://buildflutter.com/functional-model-view-update-architecture-for-flutter/`.

41 Manuel Serrano and Vincent Prunet. A glimpse of Hopjs. In *ICFP*, pages 180–192. ACM, 2016.

42 The PureScript Contributors. PureScript, 2019. Accessed on 2019-09-02. URL: `http://www.purescript.org/`.

43 Vasco T. Vasconcelos. Fundamentals of session types. *Inf. Comput.*, 217:52–70, 2012.

44 Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*, page 561. North-Holland, 1990.

45 Philip Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014. `doi:10.1017/S095679681400001X`.

46 Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. Distributed system development with ScalaLoci. *PACMPL*, 2(OOPSLA):129:1–129:30, 2018.