Multi-Tier Functional Reactive Programming for the Web

Bob Reynders

Supervisors:
Prof. dr. ir. F. Piessens
Prof. dr. D. Devriese
(Vrije Universiteit Brussel)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

October 2019
Multi-Tier Functional Reactive Programming for the Web

Bob REYNDERS

Examination committee:
Em. Prof. dr. A. Bultheel, chair
Prof. dr. ir. F. Piessens, supervisor
Prof. dr. D. Devriese, supervisor
(Vrije Universiteit Brussel)
Prof. dr. ir. T. Schrijvers
Prof. dr. B. Jacobs
Dr. ir. T. Van Cutsem
(Nokia Bell Labs)
Dr. G. Radanne
(University of Freiburg)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

October 2019
Acknowledgments

The idea of working on multi-tier FRP started with my master thesis, I wanted to work on my own topic and was lucky enough to have Frank as my supervisor. He recommended Dominique as a co-supervisor and we immediately ended up with the subject of multi-tier FRP on which I was able to do research for the following six years. Frank is one of the friendliest people I know, thank you for always making time for listening to my problems and keeping me motivated!

Unlike most people, I was lucky enough to share an office with my supervisor, Dominique. Whenever I got stuck he was a shout away and would always make time to help me reach a good solution. Without his guidance you would probably not be able to read this work and the topic of multi-tier FRP would have been collecting dust somewhere as a mediocre master thesis. While I often expressed doubt and dissatisfaction over my own results, Dominique was always there with a positive attitude towards my work.

Thank you to all the members of my jury: Tom Schrijvers, Bart Jacobs, Tom Van Cutsem and Gabriel Radanne. Your comments were insightful and definitely made the text better. Special thanks to Tom Van Cutsem who proposed changes in terminology that made it much easier to explain certain concepts, something with which I was struggling for a while!

Thank you to the Research Foundation - Flanders or “Fonds Wetenschappelijk Onderzoek (FWO)” for their four years of financial support towards my research.

Thank you to my office mates: Thomas, Thomas, Kobe, Fatih, … You always listened to my constant complaining, even after its exponential increase after each rejected paper submission ☹️. Special thanks to Thomas¹ for doing most of this on his own for three years and thanks to Thomas² for taking over and adding table tennis into the mix.
I would like to thank the people at DistriNet: Neline, Victor, Jonathan, Andreas, Hans, Katrien, Annick, Jan Tobias, Job, Raoul, Stefanos, Tobias, ..., for keeping DRADS and lunch breaks interesting! Thank you to all DistriNet secretaries for all administrative help regarding conference trips!

Thank you to my friends, you supported me throughout my studies and research and were always there for me, especially when I needed it.

Finally, thank you to my parents who always tried to figure out what I was actually doing for 5 years. You always supported me and encouraged me to find my own path and study whatever I wanted for which I will always be grateful.
Abstract

Web applications are inherently distributed, and not just because their client and server counterparts run on networked systems. Web applications are written in multiple programming languages and as multiple programs: the server and client programs. In an effort to lower the complexity of the web, multi-tier programming was proposed. In multi-tier programming languages the language and its tooling give support to create web applications as a whole, one program is written in one language.

Web applications are also inherently asynchronous. On the server, they constantly process several client requests and in the browser they constantly have to react to input, be it from the user or a server. A technique that can be applied to make such programs easier to understand is functional reactive programming. A functional programming model that models an interactive program as compositions between two primitives, behaviors and events.

In this research, we apply and combine both techniques to web applications, making advances in both applying multi-tier programming to existing languages and to functional reactive programming. We show that the whole of multi-tier and functional reactive programming is greater than the sum of its parts with new properties as a result.

Multi-tier Functional Reactive Programming

Multi-tier FRP is our unification of multi-tier programming and functional reactive programming.

It tackles two problems in development, (1) the inherent distributed and (2) the asynchronous nature of the web. With our design and implementation it becomes possible to write an entire web application (both client and server) in a single language as a composition between events and behaviors. Communication between client and server tier is done through tier-crossing primitives, a low-cost abstraction with atomic propagation. They are checked at the language level and make sure client and server are compatible.
We present Gavial, a usable and realistic implementation of multi-tier FRP in Scala as a library that builds on several other techniques to become viable, several of which we describe throughout this dissertation:

**FRP DOM APIs** are key to making a usable FRP web framework. However, subtle design decisions have a big impact and certain issues do not necessarily have an ideal solution.

What should be the entry point of the program? How do we deal with recursive definitions in HTML? The user interface is expressed in terms of itself, for example, a todo-list adds rows depending on the interaction with itself. How do we interface with the DOM? The DOM is both a producer of events and a storage of state.

We look at these questions and implement a pragmatic FRP DOM API that is used in Gavial.

**Scalagna** is our multi-tier-as-a-library project for Scala. There are several potential problems with multi-tier language development. Certain languages start from scratch and have a hard time covering all library use cases of mature language ecosystems, others start from an existing language but have a high implementation cost on client compilation, etc.

We present Scalagna, a first step towards multi-tier programming for Scala. It takes advantage of ScalaJS, a mature Scala to JavaScript compiler with its own ecosystem of libraries.

**Incremental behaviors** are an additional FRP primitive. State that is incrementally modified is common in web applications, a user that interacts with a program constantly changes its state with each form of input.

In an FRP setting these states are modeled using behaviors, but traditional FRP exposes no information on why or when these behaviors change. This information becomes crucial in a multi-tier FRP setting, where incremental data structures can become large (e.g., through multiple user interaction). Such large data structures are required to be processed efficiently, possibly in terms of computation but definitely in terms of bandwidth. To make this information available in multi-tier FRP, we present incremental behaviors. A behavior that describes why and when it changes its value so that they can be sent across the network efficiently.
Beknopte samenvatting


Webapplicaties zijn ook inherent asynchroon. Aan de kant van de server moeten webapplicaties constant verzoeken van de client verwerken en in de browser moeten ze constant reageren op invoer van de gebruiker. Een techniek dat hiervoor kan toegepast worden om zo’n programma’s makkelijker te maken is functioneel reactief programmeren (FRP). Het is een functioneel programmeermodel dat een interactief programma modelleert als composites tussen twee primitieven: behavior en event.

In dit onderzoek passen we beide technieken toe op webapplicaties, we maken vooruitgang in het toepassen van multitiertaaltechnieken op bestaande talen en op functioneel reactief programmeren. We tonen aan dat het geheel van multitier functioneel reactief programmeren groter is dan de som van de delen.

Multitier Functioneel Reactief Programmeren Multitier FRP is onze unificatie van multitier programmeren en functioneel reactief programmeren.

Het pakt twee problemen aan in ontwikkeling, (1) de inherent gedistribueerde en (2) de asynchrone aard van webapplicaties. Met ons design en onze implementatie is het mogelijk om een volledige webapplicatie (zowel client als server) te schrijven in een enkele taal als een compositie tussen events en behaviors. Communicatie tussen de client- en serverdelen wordt gedaan via speciale communicatie primitieven, een abstractie met een lage kost die toch nuttige eigenschappen heeft zoals atomische propagatie over het netwerk. Via deze primitieven is compatibiliteit tussen client en server gegarandeerd op taalniveau.
We stellen Gavial voor, een bruikbare en realistische implementatie van multitier FRP in Scala als een *bibliotheek*. Om bruikbaar te zijn bouwt het verder op andere technieken, verschillende die we zelf uitwerken en bespreken doorheen dit proefschrift:

*FRP DOM bibliotheken* zijn de sleutel tot het maken van een praktisch bruikbaar FRP web-raamwerk. Maar subtiele ontwerpbeslissingen hebben een grote impact en verschillende problemen hebben niet noodzakelijk een ideale oplossing.

Wat doen we met het toegangspunt van het programma? Hoe behandelen we recursieve definities in HTML? De gebruikersomgeving is vaak recursief uitgedrukt, bijvoorbeeld, een todo-lijst dat rijen toevoegt wanneer er op een “toevoegen” knop wordt gedrukt. Hoe koppelen we FRP aan het “document-object-model” (DOM) van de browser? De browser is zowel een bron als opslag van data.

We bekijken al deze vragen en implementeren een pragmatische FRP DOM bibliotheek die gebruikt wordt in Gavial.

*Scalagna* is ons multitier-als-een-bibliotheek-project voor Scala. Er zijn verschillende potentiële problemen met multitiertaalontwikkeling. Sommige talen starten van niets en hebben moeite om alle bibliotheken op te bouwen die volwassen talen al aanbieden. Andere starten van een bestaande taal maar hebben een grote implementatiekost bij bijvoorbeeld het compileren naar JavaScript.

Wij stellen Scalagna voor, een eerste stap richting multitier programmeren voor Scala. Het maakt gebruik van Scala.JS, een volwassen Scala-naar-JavaScript-compiler met zijn eigen waaier aan bibliotheeken.

*Incrementele behaviors* zijn een toegevoegd FRP primitief. Incrementeel opbouwende gegevensstructuren zijn vaak voorkomend in webapplicaties. Iemand die een programma gebruikt verandert immers telkens de toestand van de applicatie bij elke vorm van invoer.

In een FRP worden deze toestanden gemodelleerd met *behaviors*, maar traditionele FRP stelt geen informatie bloot over *waarom* of *wanneer* deze *behaviors* veranderen. Deze informatie wordt cruciaal in multitier FRP. Incrementele data structuren kunnen hier groot worden (bv., door meerdere interacties van gebruikers). Zulke grote data structuren moeten efficiënt behandeld worden, mogelijk wat betreft berekeningen maar vooral wat betreft bandbreedte. Om deze informatie beschikbaar te maken in multitier FRP stellen we incrementele *behaviors* voor. Dit primitief beschrijft *waarom* en *wanneer* het van waarde verandert zodat het efficiënt over het netwerk kan worden verstuurd.
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CSS</td>
<td>Cascading Style Sheets</td>
</tr>
<tr>
<td>DBehavior</td>
<td>Discrete Behavior</td>
</tr>
<tr>
<td>DOM</td>
<td>Document Object Model</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain Specific Language</td>
</tr>
<tr>
<td>EDSL</td>
<td>Embedded Domain Specific Language</td>
</tr>
<tr>
<td>FFI</td>
<td>Foreign Function Interface</td>
</tr>
<tr>
<td>FRP</td>
<td>Functional Reactive Programming</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HTML</td>
<td>Hypertext Markup Language</td>
</tr>
<tr>
<td>IBehavior</td>
<td>Incremental Behavior</td>
</tr>
<tr>
<td>IC</td>
<td>Incremental Computing</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>LMS</td>
<td>Lightweight Modular Staging</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Call</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
</tbody>
</table>
# Contents

Abstract iii  
Beknopte samenvatting v  
List of Abbreviations vii  
Contents ix  
List of Figures xv  
List of Tables xix  

1 Introduction 1  
1.1 Web Application Development 1  
1.1.1 Multi-tier Programming Languages 4  
1.1.2 Functional Reactive Programming (FRP) 6  
1.1.3 Scala as a Language for the Web 8  
1.1.4 Multi-tier Functional Reactive Programming: Research Questions 9  
1.2 Contributions 9  
1.2.1 Summary 14  
1.3 Other Research 15
2 Functional Reactive and Multi-tier Programming: A First Design

2.1 Introduction

2.2 Multi-tier FRP by Example
   2.2.1 FRP on the Client
   2.2.2 Alternative: Multi-tier FRP

2.3 Background
   2.3.1 Functional Reactive Programming
   2.3.2 JS-Scala: writing JavaScript in Scala

2.4 A Scala API for multi-tier FRP
   2.4.1 FRP API on the server
   2.4.2 FRP API for the Client
   2.4.3 Client-Server Interaction
   2.4.4 Distinguishing Clients in the API

2.5 Implementation
   2.5.1 Client & Server FRP API
   2.5.2 Basic Client & Server Interaction
   2.5.3 Client Aware Interaction

2.6 Conclusion

2.7 Future Work

3 Scalagna: towards a Multi-tier Scala

3.1 Introduction

3.2 Scalagna by Example
   3.2.1 Pages & Annotations
   3.2.2 Exposing Server State
   3.2.3 Dynamic Client/Server Interaction
CONTENTS

3.3 API .................................................................................. 52
3.4 Implementation ................................................................. 54
  3.4.1 Structure ................................................................. 55
  3.4.2 Annotations .............................................................. 56
3.5 Evaluation ........................................................................ 56

4 Functional Reactive Programming and the DOM ................. 61
  4.1 Introduction .................................................................... 62
    4.1.1 Outline ............................................................... 64
  4.2 Functional Reactive Programming .................................... 64
  4.3 Type of Main? ............................................................... 66
    4.3.1 Html. ....................................................................... 67
    4.3.2 Behavior of Html. ................................................... 68
  4.4 Recursion between FRP and the DOM ......................... 69
  4.5 Pull-based Interfaces to the DOM ................................... 71
    4.5.1 DOM Properties as Continuous Behaviors ............. 72
  4.6 Our FRP DOM Library by Example .............................. 74
    4.6.1 API ....................................................................... 74
    4.6.2 Discussion ........................................................... 76
  4.7 Conclusion ..................................................................... 76

5 Efficient Functional Reactive Programming through Incremental
Behaviors .............................................................................. 77
  5.1 Introduction .................................................................... 78
    5.1.1 Contributions ........................................................ 80
  5.2 Incremental Behaviors .................................................... 80
    5.2.1 Functional Reactive Programming: Event & Behavior 81
    5.2.2 Motivating Example: Todo List ............................. 82
6.4.3 Crossing Tiers with Incremental Behaviors . . . . . . . . . . 122
6.5 Conclusion and Future Work . . . . . . . . . . . . . . . . . . . . . . 123

7 Related Work 125
7.1 Multi-tier Reactive Programming Languages . . . . . . . . . . . . . 125
  7.1.1 Local Reactive Programming in Multi-tier Languages . . . . . 127
  7.1.2 Multi-tier Reactivity . . . . . . . . . . . . . . . . . . . . . . . . . 127
7.2 Multi-tier Languages and their Implementation: Comparing to Scalagna 130
7.3 Reactive Programming for the Client Side Web . . . . . . . . . . . . . 131
7.4 Incremental Behaviors . . . . . . . . . . . . . . . . . . . . . . . . . . 132

8 Conclusion 135
8.1 Multi-tier Scala . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 135
8.2 Efficient FRP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 136
8.3 FRP and (Multi-tier) Web Development . . . . . . . . . . . . . . . . . 137
8.4 Potential Directions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 137

A Denotational Semantics 139
A.1 Domain . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 139
A.2 Core Operations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 143
  A.2.1 Client Information . . . . . . . . . . . . . . . . . . . . . . . . . . 143
  A.2.2 Events . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 144
  A.2.3 Behavior . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 148
  A.2.4 Incremental Behavior . . . . . . . . . . . . . . . . . . . . . . . . 150
  A.2.5 Discrete Behavior . . . . . . . . . . . . . . . . . . . . . . . . . . 153
A.3 Conversion Primitives . . . . . . . . . . . . . . . . . . . . . . . . . . . 154
  A.3.1 Events . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 155
  A.3.2 Behavior . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 156
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The architecture of a web application</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Typical implementation of a shared counter</td>
<td>3</td>
</tr>
<tr>
<td>1.3</td>
<td>Architecture of a multi-tier web application</td>
<td>4</td>
</tr>
<tr>
<td>1.4</td>
<td>Multi-tier implementation of a shared counter</td>
<td>5</td>
</tr>
<tr>
<td>1.5</td>
<td>Event loop working through the callback queue and modifying state</td>
<td>6</td>
</tr>
<tr>
<td>1.6</td>
<td>The dataflow nature of FRP</td>
<td>7</td>
</tr>
<tr>
<td>1.7</td>
<td>FRP implementation of a counter</td>
<td>7</td>
</tr>
<tr>
<td>1.8</td>
<td>Simplified Multi-tier FRP</td>
<td>10</td>
</tr>
<tr>
<td>1.9</td>
<td>$x &lt; y$ over the network</td>
<td>11</td>
</tr>
<tr>
<td>2.1</td>
<td>Screenshot of our example chat application in its simplest form</td>
<td>24</td>
</tr>
<tr>
<td>2.2</td>
<td>Mental model of data flow in chat application</td>
<td>24</td>
</tr>
<tr>
<td>2.3</td>
<td>Single-tier FRP client for a chat application, written in JavaScript</td>
<td>25</td>
</tr>
<tr>
<td>2.4</td>
<td>Mental model of data flow in first version of our chat application (using client-side FRP)</td>
<td>27</td>
</tr>
<tr>
<td>2.5</td>
<td>Example chat application implemented using multi-tier FRP for the web</td>
<td>29</td>
</tr>
<tr>
<td>2.6</td>
<td>Server FRP API</td>
<td>33</td>
</tr>
<tr>
<td>2.7</td>
<td>Client FRP API</td>
<td>34</td>
</tr>
<tr>
<td>2.8</td>
<td>Multi-tier FRP chat application with private messaging UI</td>
<td>37</td>
</tr>
<tr>
<td>2.9</td>
<td>Multi-tier FRP chat application with private messaging</td>
<td>38</td>
</tr>
<tr>
<td>Figure</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>3.1</td>
<td>Visitor Counter</td>
<td>50</td>
</tr>
<tr>
<td>3.2</td>
<td>Visitor Counter Prints</td>
<td>51</td>
</tr>
<tr>
<td>3.3</td>
<td>File Read</td>
<td>51</td>
</tr>
<tr>
<td>3.4</td>
<td>Scalagna Structure</td>
<td>55</td>
</tr>
<tr>
<td>3.5</td>
<td>Shopping List</td>
<td>57</td>
</tr>
<tr>
<td>3.6</td>
<td>Eliom Draw</td>
<td>59</td>
</tr>
<tr>
<td>3.7</td>
<td>Scalagna Draw</td>
<td>59</td>
</tr>
<tr>
<td>4.1</td>
<td>To-do List</td>
<td>64</td>
</tr>
<tr>
<td>4.2</td>
<td>Event &amp; Behavior API</td>
<td>65</td>
</tr>
<tr>
<td>4.3</td>
<td>FRP DOM To-do List Manager</td>
<td>75</td>
</tr>
<tr>
<td>4.4</td>
<td>FRP DOM API</td>
<td>75</td>
</tr>
<tr>
<td>5.1</td>
<td>Event &amp; Behavior API</td>
<td>81</td>
</tr>
<tr>
<td>5.2</td>
<td>FRP Todo List Example</td>
<td>83</td>
</tr>
<tr>
<td>5.3</td>
<td>FRP Incremental Todo List Example</td>
<td>85</td>
</tr>
<tr>
<td>5.4</td>
<td>Reactive Sequence Core API</td>
<td>87</td>
</tr>
<tr>
<td>5.5</td>
<td>Reactive Sequence Implementation</td>
<td>88</td>
</tr>
<tr>
<td>5.6</td>
<td>Map Implementation</td>
<td>89</td>
</tr>
<tr>
<td>5.7</td>
<td>Updating mapped vectors</td>
<td>92</td>
</tr>
<tr>
<td>5.8</td>
<td>Updating logged vectors</td>
<td>92</td>
</tr>
<tr>
<td>6.1</td>
<td>CircleRoyale</td>
<td>97</td>
</tr>
<tr>
<td>6.2</td>
<td>FRP Primitives</td>
<td>99</td>
</tr>
<tr>
<td>6.3</td>
<td>The tiers available in Gavial.</td>
<td>103</td>
</tr>
<tr>
<td>6.4</td>
<td>$x &lt; y$ across tiers</td>
<td>114</td>
</tr>
<tr>
<td>6.5</td>
<td>Structure of the Gavial implementation.</td>
<td>121</td>
</tr>
<tr>
<td>6.6</td>
<td>Auto Completion for Gavial in IntelliJ</td>
<td>121</td>
</tr>
</tbody>
</table>
6.7 Efficient Tiered Glitch Freedom .............................. 122
List of Tables

3.1 Server Performance (ms) ........................................ 57
3.2 Client Performance (ms) ........................................ 58
7.1 Comparison Table for Multi-tier Reactivity ................. 126
1 | Introduction

The web and its applications have come a long way since the first introduction of HTML. Over the years simple websites evolved from static information pages to full-fledged applications that have access to location data, local storage, etc. Web applications are no longer simple one page programs.

In this dissertation we look at a programming model for modern web applications, multi-tier functional reactive programming. The main contributions of this thesis are presented in the following chapters based on the peer-reviewed publications of these contributions. In this first chapter we put our contributions in context. In section 1.1, we discuss multi-tier programming, functional reactive programming and Scala as a language for web applications. We show how this work is the foundation of our multi-tier functional reactive programming research. In section 1.2, we show our work and discuss our contributions to three fields: multi-tier functional reactive programming, functional reactive programming and multi-tier programming. In addition to these contributions, other research I have worked on is summarized in section 1.3. In section 1.4, we discuss the remaining chapters of the thesis. The remaining chapters contain in-depth information about the contributions. Each chapter corresponds to an original unmodified peer-reviewed (or in review) paper.

1.1 Web Application Development

Websites were simple when the web was first introduced. Static information was encoded in simple HTML files. Markup was later added with CSS, a simple language meant to make it easy to express certain simple presentation properties such as adding colors. As websites got more popular, the simple static pages evolved towards more scalable solutions. Server-side programs started generating HTML with information stored in databases (using SQL) and websites became actual programs. Not just on the server, but also on the browser. JavaScript made it possible to write interactive HTML pages.
Features were added rapidly and the web evolved into the most popular application platform. Nowadays, typical web development requires several programming languages (HTML, CSS, JavaScript, etc.) and a complete web application is written as a combination of at least two programs, a client and a server program. As shown in fig. 1.1, static information is still represented in HTML, markup is added with CSS, interactive client features are programmed in JavaScript and this application communicates with a server program written in a server language of choice (e.g., PHP) which in turn gets data from a database using a query language such as SQL. At one side, users are expected to be able to interact with the program at all times, while the web server is expected to respond to as many client programs as possible. Both client and server programs are highly reactive environments that constantly deal with asynchronous events.

Let us go through the simple example shown in fig. 1.2. It is a shared counter that can be increased by pressing a button. Updates to the counter are made visible across all clients.

Two distinct programs are shown, one deals with interface updates and one deals with counting. The client program consists of some HTML (served by the server) and two JavaScript functions. When the page is loaded a callback is immediately registered on an event source, an abstraction that allows JavaScript programs to subscribe to server updates on the given URL. Whenever the server publishes an update, the interface is changed to reflect the new state of the shared counter. The second JavaScript function is bound to the button’s click event and sends increment requests to the server. The server program provides two services, one simply increments a variable while the other pushes updates to all subscribed clients.

Even in this simple example a programmer has to write two inter-operable programs that react to multiple events.
Listing 1.1: client.js

```javascript
function add() {
  const xhr = new XMLHttpRequest;
  xhr.open('GET', '/increment');
  xhr.send();
}

EventSource('/update').onmessage = function (m) {
  document.querySelector('button').innerHTML = m.value;
};
```

Listing 1.2: server.scala

```scala
var state = 0
val (sseActor, sseSource) = Source.actorRef[ServerSentEvent]().
toMat(BroadcastHub.sink)(Keep.both).run()
def sse(int: Int): ServerSentEvent = ...

val route = path("/") {
  get("index.html") {
    complete("<html><script src='client.js'></script>/<script/>
    <body><button onclick='add'>" + state + "</button></body></html>"
  }
  get("increment") {
    complete {
      state += 1
      sseActor ! sse(state)
    }
  }
  get("updates")(complete(sseSource))
}
```

Figure 1.2: Typical implementation of a shared counter
1.1.1 Multi-tier Programming Languages

A web application requires multiple programming languages (up to 5\textsuperscript{1}) and typical web development languages such as JavaScript, HTML and CSS are still constantly evolving. Industry JavaScript frameworks are adding support for HTML syntax while CSS is getting support for variables and calculations. In other words, features of certain languages seem to be desired in others.

Neubauer and Thiemann [2005], Serrano et al. [2006], Cooper et al. [2007] proposed multi-tier programming, in which a programmer uses a single language to write web applications as single programs to minimize overhead. The programming language provides support to the programmer to write both client and server sections.

Contrary to traditional web development, multi-tier web development treats web applications as a single program. It provides developers with extra support to make sure that both client and server sections of the program are compatible and makes it possible to write abstractions that transcend a single tier. As shown in fig. 1.3, the compiler or toolchain of the multi-tier language generates the normal web development layout. How far a multi-tier language takes this idea is dependent on the language, e.g., Chlipala [2015] proposes Ur/Web, a language with typed HTML, CSS, SQL and client/server abstractions.

Let’s look at an example in Scalagna, a multi-tier programming language that we made as part of this thesis (based on existing Scala compilers). In fig. 1.4, we implement the same shared counter example as before.

Scalagna provides means to create services, similar to those in Eliom [Radanne et al., 2016], they are the entry points to the web server. We make use of three different languages: JavaScript, CSS, HTML and a server language such as PHP.

\textsuperscript{1}HTML, CSS, SQL, JavaScript and a server language such as PHP.
services in our example, an HTML service used to serve the HTML application, an SSE service used to implement the broadcasting from server to clients and an RPC service used to call a server function from the client.

The rest of the application logic has remained exactly the same and has been purposely written to match the previous example. Note several important differences. The program is no longer split across different programming languages and is written as one Scala program. Interfacing between client and server sections is completely supported by the language. There are no longer two paths for each client/server connection to be kept up-to-date (which the attentive reader might have noticed already caused a bug in the original example due to a typo in the update URL). The broadcasts are now typed, `sseUpdates` only works for integers and its methods `push` and `onmessage` are typed accordingly.

In short, it is possible to write high-level abstractions in multi-tier languages that would otherwise not be possible. It gives programmers tooling such as typechecking and auto-completion for tier-crossing behavior.
1.1.2 Functional Reactive Programming (FRP)

The complexity of web applications does not just come from its distributed nature (with regard to both infrastructure and languages). A web application inherently deals with many events. For example, the client side of a web application reacts to communication from the server as well as users pressing keyboard buttons or using the mouse to interact with the DOM. The single-threaded JavaScript programming model for browsers deals with such events using callbacks, visually shown in Fig. 1.5. An event loop continually clears a queue of events. Each event executes its assigned callbacks which in turn modify the state of the program. Programmers write these callback functions to react to event data and attach them to events of interest, e.g., a specific button being clicked.

This callback-style of programming is not just prominent on the client, but on the server program as well, for example to respond to requests from the client. A problem with this style of programming is that it is a tedious task to figure out the dependency graph between events, that is, to know what exactly changes in the program when certain events fire. Developers reading the code have to piece together the control flow that is scattered in the program through (possibly nested) callbacks and mutable variables, a phenomenon and code-smell named "callback hell". This makes it hard to figure out why and where state changes take place. A programmer has to scan through the entire section of code that has access to the mutable variable. Callbacks (which are just functions called in response to events) do not compose. To somehow react to multiple events at once programmers have to resort to mutable state and manual scheduling of functions. For an extensive list of software engineering principles violated by the callback (or observer) style of programming we like to refer to Maier et al. [2010].

An alternative to writing programs that react to events proposed by Elliott and Hudak [1997] is functional reactive programming (FRP). FRP applications are written as compositions between two primitives: behaviors and events. Events are first-class values that represent a stream of events. For example, the abstraction to model clicks on a button would be an event of click data, i.e., Event[Click]. Behaviors on the other hand model time-varying values, they are always defined and can change value. For example, the FRP equivalent of the value of an input box would be a behavior of
val eventSource = EventSource[Any]()
val increments: Event[Int] = eventSource.map(_ ⇒ 1)
val count: Behavior[Int] = increments.fold(0)(_ + _)
val ui: Behavior[HTML] = count.map { c ⇒
  button(onclick := eventSource, c)
}

strings: Behavior[String]. Its value would be the empty string for an empty input box and would change accordingly. There are primitives that make it possible to compose events and behaviors and the FRP primitives themselves can be transformed using typical functional programming primitives such as map. In FRP, events and time-varying values are first-class and in contrast to callbacks they are made to be composed and their interdependencies are explicit and easy to track in codebases.

FRP programs can be thought of as dataflow programs, as shown in fig. 1.6. The “world” pushes event data into the data flow which then affects the world in return.

Similar to the previous example, we implement a counter using an FRP style in fig. 1.7. To simplify, we use HTML to describe the interface but no longer implement a server, it is a simple counter.

In this example, the application is defined as a behavior of html element ui, that is, an html element that changes its value throughout time. ui is defined in terms of count which is a simple fold over the program’s increments. The behavior starts at value 0 and continually adds new events to its accumulator. These events are defined as increments, an event created by replacing the on click values in eventSource with the integer ‘1’.

Figure 1.6: The dataflow nature of FRP

Figure 1.7: FRP implementation of a counter
1.1.3 Scala as a Language for the Web

Throughout the dissertation we often use code examples to showcase the designed APIs and give a hands-on feeling. Apart from comparisons to related work, we always use the Scala language in this thesis. Scala was introduced in 2004 by Odersky et al. and steadily became popular as a JVM language for mainly server-side development with adoption by companies such as Twitter, LinkedIn, etc.

However, we consistently use Scala for all tiers of the application and Scala has become a language for all parts of a web application. Let us highlight the projects that make it possible to use Scala as an alternative to JavaScript for web applications.

**JS-Scala**  
JS-Scala [Kossakowski et al., 2012] is used in chapter 2, it is an embedded domain specific language based on lightweight modular staging (LMS [Rompf and Odersky, 2010]) that allows programmers to write staged expressions in Scala. These staged expressions (noted by their type: \texttt{Rep[T]}) are generated as JavaScript at run-time. In short, it makes it possible for Scala programs on the JVM to generate JavaScript.

Lifting expressions from Scala to JS-Scala is possible in specific cases but transformations have to be defined for a type \texttt{T} to \texttt{Rep[T]}. Implications are that re-using functions such as \( A \Rightarrow B \) in the staged language or vice versa becomes more difficult (i.e., it is hard or impossible to transform \( A \Rightarrow B \) to \texttt{Rep[A \Rightarrow B]}). Other limitations are the lack of JavaScript support for regular Scala objects, i.e., creating a class \texttt{C} and having out-of-the-box support for meaningful \texttt{Rep[C]} expressions. JS-Scala has to limit which classes are supported, leaving programmers quite restricted.

Libraries for JS-Scala are regular Scala libraries. They provide functions that work with the staged (\texttt{Rep[T]}) expressions of the language and can be organized in regular Scala classes or objects.

**Scala.JS**  
Scala.JS [Doeraene et al., 2016] on the other hand is a mature Scala to JavaScript compiler implemented as a compiler plugin. It takes existing Scala programs and compiles them to JavaScript. Programmers are able to use all features\(^2\) of Scala (including the standard library) on the JavaScript platform.

The existing Scala standard library is based on the Java standard library. To make it easy for Scala programmers to make their Scala applications compatible with Scala.JS, parts of the Java standard library have been re-implemented in Scala. Thus, Scala.JS compiles its programs against a ported (API compatible) version of the standard library.

\(^2\)With the exception of low-level APIs that are JVM specific, e.g., low-level threading APIs.
Scala libraries that do not contain JVM or platform specific code can be cross-compiled with Scala.JS as-is. This split of libraries ties into the ecosystem through tool support. The Scala build tool separates normal JVM libraries from Scala.JS libraries. Implications are that Scala.JS programs compile against, and produce, different “JavaScript specific” libraries.

1.1.4 Multi-tier Functional Reactive Programming: Research Questions

While functional reactive programming has been applied to web applications or in limited ways to multi-tier programming, the focus has always been to make client-side programming easier. In this dissertation we aim to unify functional reactive programming and multi-tier programming completely and we investigate the question: “can a web application be written just by composing functional reactive programming primitives?”. The goal is to investigate whether or not it is possible to write a complete web application in a programming style that makes dependencies between events and event-driven state explicit and easy to track down.

We do not consider this answer to be solved by just providing a semantics or a prototype implementation. Beyond providing multi-tier FRP primitives, we aim to build a realistic multi-tier FRP web framework that works as shown in fig. 1.8 where a functional reactive program gets compiled to JavaScript, HTML and the JVM through Scala.JS and Scala. To do so we identify several hurdles that have to be overcome related to multi-tier functional reactive programming:

- Can we make multi-tier FRP efficient in terms of network bandwidth?
- Can we design multi-tier FRP such that tooling and libraries of a base language can be reused?
- Can FRP primitives be given an intuitive semantics that takes into account essential distributed system aspects such as network delay?

1.2 Contributions

In this section we summarize our contributions in the fields of multi-tier programming in Scala, functional reactive programming and multi-tier functional reactive programming all leading up to a usable web framework in which applications are created through (cross-tier) composition of FRP primitives.
Multi-tier Functional Reactive Programming

The biggest and most important contribution of this work is Gavial, a mature implementation and design of multi-tier functional reactive programming in Scala.

While FRP has already been applied quite often to client-side web applications to further help deal with events — it has not been used to model the server. However, both parts of a web application frequently deal with and respond to events. FRP can be applied in both the client and the server parts of a web application in a multi-tier language. We propose our design with denotational semantics (available in appendix A) and an implementation at https://github.com/tzbob/gavial.

Note that Gavial is the end result of all research in this dissertation, but there are two distinct chapters on multi-tier functional reactive programming. An initial design and prototype in chapter 2, which shows the potential and discusses some immediate benefits and future work and the final design in chapter 6 which discusses new techniques to make multi-tier functional reactive programming feasible and better.

Gavial has a number of novel and desirable features or properties such as tiered glitch freedom, bootstrapping, a flexible 3-tiered system and an FRP DOM API.

Tiered Glitch Freedom (Ch. 6)  Developers no longer write their web application by imperatively sending data from client to server and vice versa. Our FRP tier-crossing primitives strike a careful compromise between a nice and abstract semantics (that allows for easy understanding and manipulation of programs) and efficient implementability on the web. Under the hood there is no magic or extensive middleware, the primitives are a thin layer on top of existing web technology and
do not require complicated distributed algorithms. Despite being a lightweight implementation it gives programmers extra guarantees that they do not typically have in current web frameworks. We support something we call “tiered glitch freedom”.

In the following example:

```scala
val x: Behavior[Int] = -- cut --
val y: Behavior[Int] = x.map(_ + 1)
val t: Behavior[Int] = x.map2(y)(_ < _) // true
```

We would expect `t` to always evaluate to true no matter how values are propagated in `x` or `y`. If such partial updates can be observed, they are called glitches. Our proposed multi-tier FRP has a similar property even in the existence of network-crossing primitives. To explain this, we visualize propagation of the above toy example as a graph in fig. 1.9 and add a network between `x` and `y`, and `t`.

![Figure 1.9: x < y over the network](image)

Even in this case, `t` always remains true, no matter the delays on the network. This tiered glitch freedom property holds as long as an expression’s dependencies reside within the same tier. Developers can take advantage of this feature since they no longer need to resynchronize manually between dependent parts of the program, FRP does it for them.³

**Bootstrapping a Web Application (Ch. 2)** Our multi-tier FRP proposal solves “bootstrapping”, a problem all web developers face and one that has many suboptimal solutions, that is, inserting the “current” state of values into the page as the client first loads the program. Typically, developers manually have to define what defines the default state of a web page. They write code server-side that imperatively inserts “current” values into the generated HTML, these values then represent the state of the program. Bootstrapping is inherent to multi-tier FRP, developers design their program by composing FRP primitives to eventually create a client html behavior that can be rendered. The semantics define these behaviors to inherently encode

³It is possible to manually break this consistency guarantee using an asynchronous FRP primitive.
bootstrapping, that is, the tier-crossing primitives that convert values from one tier to the other define the value at the time of opening the webpage. Developers no longer need to define this behavior manually, for example, a chat application written in multi-tier FRP inherently contains whatever is defined as the “current” state on the server.

A 3-Tiered System for the Modern Web (Ch. 6) We propose a 3-tier system for client/server web applications. A client tier and two server tiers: session and application. Both client and session tiers consist of computations belonging to a specific client (client and server side respectively). The application tier contains computations catering to one or more clients. This multiplicity of multiple client/session tiers with one application tier shows up in the API.

The purpose of this 3 tier system is to make several types of applications natural to write in our framework. In traditional server web frameworks programmers write code from the perspective of handling an HTTP request. Computations are in direct response to such requests. The session tier is designed for such a way of processing requests for specific clients. However, web applications are usually designed not just to provide a service for individual interaction, but also to provide a service that allows individuals to interact with each other. These type of programs are written with the application tier. A program that just stores and shows data would be written using the client and session tiers. A chat application would be written in all three, UI and other design features on the client, input validation or sanitization on the session tier and the actual chat log would be built in the application tier.

Web applications no longer use purely one-sided communication, there are use cases that require servers to update clients and this can be done in multiple ways (server-sent events, websockets, long-polling, etc.) which we support transparently in Gavial. Depending on which FRP primitives are used and how the program is composed, it runs on a websocket backend. Applications that run entirely in the session tier or that are composed in a way that can be implemented without requiring server-to-client-communication run in a regular XMLHttpRequest backend.

FRP DOM API (Ch. 4) During development of a better FRP API for the DOM in our project we identified various non-trivial design problems that often did not have ideal solutions. We identify the following issues and discuss the implications of possible solutions.

What should be the entry point of the program? In FRP, a program can be defined as a single value made up of events and behaviors. But, what should the type of that single value be? Do HTML elements contain time-varying values
(element containing Behaviors) or is the program itself time-varying (Behavior of Element)?

How do we deal with recursive definitions in HTML? The user interface is expressed in terms of itself, for example, a todo-list adds rows depending on the interaction with itself.

How do we interface with the DOM? The DOM is both a producer of events and a storage of state. We look at trade-offs in APIs that use Events to model DOM events and Behaviors to model DOM state.

In summary, regarding multi-tier functional reactive programming, we provide:

- A design and implementation of multi-tier FRP specified with denotational semantics and implemented as a library in Scala.
- The tier-crossing primitives have a form of glitch minimization across network communication that gives both useful guarantees as well as being low-cost in implementation and runtime overhead.
- Support for both websocket as well as regular XMLHttpRequest backends to multi-tier FRP.
- A pragmatic FRP DOM API that allows developers to model and read the DOM.

In support of the development of Gavial, we made a number of contributions that also have independent value: Scalagna and incremental behaviors.

Multi-tier Scala as a Library

We designed and developed Scalagna, a first prototype of Scala as a multi-tier language by compiling codebases twice. In Scalagna a program is compiled once with the Scala compiler and once with the Scala.JS (Scala to JavaScript) compiler. Scalagna is implemented as a library and provides developers with @client and @server annotations which they can use to write multi-tiered code. This results in a simple yet effective approach to a multi-tier language since there were already ecosystems in place for client (Scala.JS) and server (Scala) web development with existing development environments that remain supported.

We provide:

- A design to create a multi-tier language without compiler modifications through compiling codebases twice.
• A prototype implementation as a library of a multi-tier Scala that is compatible with both the Scala ecosystem of libraries as well as the Scala.JS ecosystem.

Efficient FRP with Incremental Behaviors

Chapter 5

The original FRP paper by Elliott and Hudak [1997] uses animations as its canonical example and purpose. FRP was later used to model user interfaces and even event based systems in general. In this dissertation we go one step further and use it to model both client and server side functionality. In this use-case there is one recurring application pattern that is not common in the animation world and that is incrementally built state. State of the program that incrementally changes — and especially grows — in response to handling events. For example, a chat application has conversations between clients as its state. Whenever a client submits a message to another user, the state is built up. However, in FRP there is no way to describe an incrementally time-varying value. In practice, developers would have to express time-varying values with events and manually model incremental state efficiently. This is particularly harmful for performance or readability in a web setting where incremental values also have to be sent across the network. We want programmers to be able to write multi-tier FRP programs that are both efficient and natural to express.

We provide:

• Incremental behaviors, a general and lightweight additional FRP primitive that encapsulates how and when its value changes. We show it is a general concept by implementing existing work on top of it such as an incremental collection API.

• An implementation of an FRP Scala library containing incremental behaviors and incremental collections.

1.2.1 Summary

To summarize the main contributions of this thesis, we propose:

• Multi-tier functional reactive programming in chapters 2 and 6, a composable technique to writing web applications as a single FRP program and show that it solves existing issues such as bootstrapping a web application while empowering developers with tools such as a three-tiered system and minimal glitch replication.

• Gavial in chapter 6, an implementation of multi-tier FRP based on Scala and Scala.JS
• A **pragmatic FRP DOM API** in chapter 4 while discussing and identifying FRP DOM issues.

• **Scalagna** in chapter 3, a first step towards a design and implementation for multi-tier Scala as library.

• **Incremental behaviors** in chapter 5, a new native FRP primitive that expresses incremental operations that can be used as a more general concept upon which existing work can be implemented.

### 1.3 Other Research

The work in this dissertation is a collection of research that most directly impacted multi-tier functional reactive programming. In the past four years we did other research on multi-tier programming in general and on combining typed EDSLs with other typed environments. We do not record this research in this dissertation but give a small summary nonetheless.

**Multi-tier Programming and Debugging with a Functional Language** Other than developing Scala-related multi-tier frameworks, we also worked on an Elm multi-tier language. For this project we collaborated with Jeff Horemans during his master thesis. In this research we created a multi-tier language based on Elm [Czaplicki, 2012]. It was based on ideas similar to Haste [Ekblad and Claessen, 2015], the project is compiled twice. Once where effects annotated to be run on the server are ignored and once where they are run, vice versa for the client. We built a visual timeline debugger on top of this multi-tier language. In the debugger both tiers of the program are visible and a timeline of all state changes is shown. This is just a small case study of where multi-tier programming can deliver better development environments, we believe much more can be done.

Publication:

Jeff Horemans, Bob Reynders, Dominique Devriese, and Frank Piessens.
In TFP, 79–97.

While the initial idea was my own, the project was fleshed out and implemented by Jeff Horemans. The paper was also written by Jeff with revisions by Frank Piessens, Dominique Devriese and me.
Multi-tier Functional Reactive Programming for IoT  We developed an EDSL for which we used functional reactive programming as a programming model for IoT devices. The language allows developers to write modules with explicit entry and exit points in the form of events. Modules are compiled to annotated C in order to run on Sancus [Noorman et al., 2013], a secure module architecture. FRP primitives are carefully compiled away to a single C loop in order to reduce overhead. This work was done in collaboration with Ben Calus for his master thesis.

Publication:


Safe Boundary APIs between Typed EDSLs and Typed Environments  In our first multi-tier implementation we used a Scala EDSL for Javascript and communicated with a library that was implemented in Scala.JS. In this research we safely generate an API for an existing library from a typed environment (Scala to Scala.JS) in a typed EDSL (JS-Scala). We model the typed environment in the EDSL and generate types accordingly.

Publication:


The idea, implementation and paper were done by me with remarks from Dominique Devriese and Frank Piessens.

1.4 Outline

The rest of the dissertation contains in-depth information regarding all contributions. Each chapter is a slightly modified version of a previous paper of which I was the first author. We start each chapter with a small introduction while the rest contains the following minor changes compared to the papers:
• Fixed several minor errors such as typos or outdated APIs.
• Changed terminology of “replication” primitives to “tier-crossing” primitives
• Grouped related work sections of all papers and made a summary in Chapter 7.

Chapter 2 describes the initial design of multi-tier FRP. It proposes a two-tier system and highlights all advantages that are gained from simply combining multi-tier programming with FRP. Server code is Scala while client code is generated through JS-Scala.

Chapter 3 describes a design and implementation for a multi-tier Scala as a library based on the two compilers: Scala and Scala.JS.

Chapter 4 discusses a DOM API for FRP. It identifies certain design issues and provides a pragmatic design and implementation.

Chapter 5 proposes incremental behaviors. An additional FRP primitive that makes it possible to express incremental computations.

Chapter 6 propose Gavial. It integrates all previous work and improves upon chapter 2. It provides solutions to issues that existed and introduces a three-tier system that make multi-tier FRP suitable for more web applications. It proposes a design as well as an implementation.

Chapter 8 concludes the thesis by looking back at how the contributions were achieved and what could be done in the future.

Appendix A contains the denotational semantics for the core of Gavial, the three-tiered multi-tier functional reactive programming framework.
In this chapter we look at our initial effort of combining functional reactive programming (FRP) and multi-tier programming. We describe a two-tiered system (client and server) where it is possible to write simple client/server applications by composing FRP primitives. Chapter 6 describes the final version of the tier-crossing primitives and features a three-tiered system instead.

This chapter is based on the following publication:


The research was performed by me with guidance from Dominique Devriese and Frank Piessens. The contents of the paper were written by me and rewritten and revised by Dominique Devriese with feedback from Frank Piessens.

Abstract

The development of robust and efficient interactive web applications is challenging, because developers have to deal with multiple programming languages, asynchronous events, propagating data and events between clients and servers, data consistency and much more. Several approaches for (partly) addressing these challenges have been proposed. Two relevant ones are (1) multi-tier languages and (2) functional reactive programming (FRP). Multi-tier programming languages support the development of client and server in a single language, and hide much of the complexity related to distribution. FRP offers the right abstractions to make event-driven programming convenient, safe and composable. However, existing web frameworks
and programming languages exploit the benefits of both approaches separately, for example by restricting the use of FRP to the client side.

We propose multi-tier FRP for the Web, a novel approach to writing web applications that deeply integrates FRP and multi-tier languages, and where the whole is greater than the sum of its parts. In multi-tier FRP, the developer programs server and client together as an FRP application composed of behaviors (signals) and events. He/she chooses explicitly where the boundary between server and client is crossed. To make our approach more concrete and provide evidence of its potential, we present a concrete design and implementation of a multi-tier FRP API for the web in the programming language Scala, using an embedded JavaScript DSL that makes Scala usable as a multi-tier language. This allows us to present initial evidence of the benefits of the multi-tier FRP approach on example applications, and to experiment with possible answers to the remaining questions. Concretely, we show possible solutions for problems like exposing client identity on the server and efficiently pre-loading clients with the latest application state. Our results show that multi-tier FRP is a promising, declarative, yet practical way of writing web applications.

2.1 Introduction

Developing interactive web applications presents a number of interesting challenges for programmers. One important challenge is the inherent distributed nature of the platform with parts of the application running on the server and other parts on the (zero or more) clients. Another challenge is dealing with the asynchronous communication that is inherent to user communication and typically used in the web’s client-server communication for reasons of performance, failure tolerance and responsiveness towards the user.

The standard approaches for dealing with these challenges (the use of callbacks and separate server- and client-side codebases, typically in different programming languages) present significant downsides. In recent years, interest in web application development has not ceased to increase (both in research and industry) and several novel approaches have been proposed to improve over these approaches. In this dissertation, we focus on two such novel solutions specifically: Functional Reactive Programming (FRP) and multi-tier languages.

Asynchronous communication and FRP The standard approach for dealing with asynchronous user and client-server input and output is the use of *callbacks*: imperative components that are invoked in response to asynchronous events. Specifically, web applications use JavaScript event handlers on the client side and HTTP request handlers on the server side. A web application containing many such
callbacks, all potentially modifying the application’s mutable state, may have a very complex control flow within and across both parts of the application. Such code can be very difficult to reason about.

FRP [Elliott and Hudak, 1997] (although initially proposed for modelling animations) can be used as an alternative programming model for asynchronous applications. Instead of using side-effecting callbacks, the program is constructed by composing behaviors (also known as signals) and events: components representing time-dependent values. Programs constructed in this manner can be given elegant denotational semantics, compose nicely and are relatively easy to reason about. FRP has been applied to the client-side web setting in several practical frameworks and languages like Flapjax [Meyerovich et al., 2009], Ur/Web [Chlipala, 2015] and Elm [Czaplicki, 2012, Czaplicki and Chong, 2013].

Recently, we are also seeing an increase in mainstream adoption of reactive frameworks that provide enhanced databinding: Liberty and Betts [2011]’s JavaScript implementation of Meijer’s .NET Reactive Extensions [Meijer, 2010], Google’s AngularJS [Google, 2010] and Facebook’s reactive UI framework React [Facebook, 2013]. The precise boundary of FRP appears to depend on who you ask, but for the purposes of this text, we consider reactive frameworks as related to FRP, but not quite the same. They use abstractions that are similar to FRP behaviors (i.e., time-varying values that propagate changes), but are not really pure FRP. They step outside pure FRP with e.g. support for attaching imperative callbacks or imperative (un-)subscribing to events. While these techniques have merit, they cannot be given an equally elegant denotational semantics as FRP and are harder to reason about. In this project, we work with pure FRP and although Scala does not allow us to enforce this, we do not intend the programmer to use imperative callbacks or other impure extensions.

The web as a distributed platform To deal with the web platform’s inherent distribution of an application between client and server, an application is often split into separate client-side and server-side programs. The JavaScript programming language is typically used for the client-side part, a server-side language of choice for the part on the server and often manually serialized messaging for communication between the tiers. However, this approach presents important downsides like the separation of an application over two partial codebases (often in different programming languages), the need for compatible marshalling and unmarshalling of client-server communication and the limitations of the JavaScript programming language (no static types, peculiar semantics [see e.g. Guha et al., 2010] etc.).

To deal with the distribution of web applications over client and server, the literature has recently seen the appearance of multi-tier languages [see among others Cooper et al., 2007, Serrano and Queinnec, 2010, Neubauer and Thiemann, 2005, Google,
In such languages, both the client and server parts of a web application are written in a single codebase and a single programming language. From this single codebase, the language implementation produces client-side executable code (typically in JavaScript) and interprets or compiles it for execution on the server. Often, the language provides synchronous and/or asynchronous communication primitives without requiring the programmer to write (un)marshalling code for the messages. The advantages of multi-tier languages are that web applications are no longer separated over separate codebases in separate programming languages, the additional features that the single language may offer over JavaScript (e.g. a static type system) and native support for client-server communication. Note that in most systems it is still the programmer who delineates the client-side and server-side parts of the application by explicitly annotating where the client-server boundary is crossed.

Combining FRP and multi-tier languages FRP and multi-tier languages have been combined before in the web setting, but always by using FRP solely at the client-side [Chlipala, 2015, Bazerman, 2012]. This approach combines the advantages of both ideas, but only in a limited way. It brings the FRP model only to one part of the application and asynchronous client-server communication on the server is treated differently (using callbacks) than other asynchronous input/output (using FRP).

In this chapter, we propose to integrate FRP and multi-tier languages more deeply by applying what we call multi-tier FRP. The idea is to construct the entire program as a single FRP application. Client- and server-side behaviors and event streams are statically distinguished and the boundary can only be crossed explicitly. This is done using new primitives that send a client-side behavior or event stream to the server or vice versa.

To make our proposal more concrete and to present evidence of its potential, we present a concrete instantiation of our approach in the form of a detailed API design and implementation in the programming language Scala, using an embedded JavaScript DSL that makes Scala usable as a multi-tier language. The precise definition of the primitives is tightly coupled to the characteristics of the distributed web platform, like the fact that there is one server but multiple clients, and the fact that the clients need to be distinguishable by the application.

A very special characteristic of the web platform that influences the primitives is the fact that web clients may be created or destroyed at any time and that the client code is not pre-installed on the client device but provided by the server when it is started. This allows the server to always deliver the latest version of the client-side

---

1See Section 2.7 for some ideas about dealing with server replication.
code and even adapt it just before being sent. This characteristic of the web platform turns up in our API for sending server-side behaviors to the client. When a user opens an application written in our framework, client-side code will be provided that is pre-loaded with the current state of server-side behaviors that were sent to the client-side, a feature which we have found both very useful and elegant.

For modeling JavaScript parts of the application, our Scala API and implementation make use of the JS-Scala library by Kossakowski et al. [2012], based on the Scala Lightweight Modular Staging (LMS) framework by Rompf and Odersky [2010]. The use of JS-Scala allows us to treat JavaScript as an Embedded Domain Specific Language (EDSL) from within Scala. The types of our client-server APIs need to mention the LMS $\text{Rep}^T$ types to ensure that client-side parts of the application can in fact be compiled to JavaScript. We support transparent serialization of behaviors’ content types from client to server and back by requiring a serializer object to be available as a Scala implicit object [Oliveira et al., 2010].

We emphasize again that while our proposed API and our implementation are necessarily Scala-specific, our approach is more general. While we have designed the API to support a library implementation in an unmodified widely used programming language, implementing our approach in a true multi-tier language could have advantages of its own, particularly avoiding the need for $\text{Rep}^T$ types in the API.

In summary, our contributions are the following:

- We propose the novel approach of multi-tier FRP for the Web.
- We instantiate our approach with the API design and implementation of a web framework in the programming language Scala, using an embedded JavaScript DSL that makes Scala usable as a multi-tier language. We explain how the API design reflects the characteristics of the web platform, specifically the need for distinguishing clients on the server and the opportunity for efficiently pre-loading client code with the latest required application state.
- We show the feasibility and potential of our approach by demonstrating simple demo applications.
- Finally, we discuss future work: challenges for developing a full-featured web framework in our approach and some of our ideas to address them.

In the rest of this chapter, we start by introducing our proposal for multi-tier FRP for the web in Section 2.2. Next, after some more background in Section 2.3, we present our concrete Scala instantiation of the approach with the API design in Section 2.4 and the implementation in Section 2.5 while Section 2.6 contains an overall summary of our proposal. We present ideas on future work in Section 2.7 and give an overview of related work in Chapter 7.
2.2 Multi-tier FRP by Example

In this section we will present our approach informally based on a simple chat application example, before presenting details in the next sections. For presentation reasons, we first show an FRP implementation of only the client part of the chat application (with an unspecified server part) and then show our proposed approach with both tiers of the application as part of a single FRP implementation.

Our example application is very simple, although we will make it slightly more realistic later on. For now, Figure 2.1 shows a screenshot: there are two input fields (name says message), a send button and a chat log. Multiple clients can connect to the single server. Figure 2.2 shows a very simple mental model of the implementation of the application. FRP implementations are often quite close to this kind of schematic representation, as we will see further.
function Entry(name, message) {
    return { "name": name, "message": message };
}

var entry = $('#name').values
    .combine($('#msg').values, function(n, m) {
        return Entry(n, m);
    });
var clicks = $('#send').asEventStream("click");

function mkRequest(entry) { /* create POST req */ }
var requests = entry.sampledBy(clicks).map(mkRequest);
doPOST(requests, "example.com/input");

var chatSrc = new EventSource("example.com/output");
var chat = chatSrc.asEventStream("message")
    .map(function(e) {
        return JSON.parse(e.data);
    });
function template(entries) { /* convert to html */ }
var view = chat.map(template);
render(view);

Figure 2.3: Single-tier FRP client for a chat application, written in JavaScript.

2.2.1 FRP on the Client

Figure 2.3 shows a JavaScript implementation of the client-side part of our chat application. The code uses the JavaScript FRP library BaconJS [Paananen, 2012] with some convenient extensions of our own that we don’t discuss in detail. Some functions that are not important for the presentation are left unspecified for brevity. It serves as a demonstration of how a simple chat client would be modeled if functional reactive programming is used on the client side. Let us take a closer look at the implementation.

The code defines a number of behaviors and event streams. These are core FRP abstractions that we will introduce properly in Section 2.3.1. A behavior represents a value in the application that may change over time, such as the contents of the chat log or the contents of a user input field. An event stream represents a channel on which new values appear at certain times, such as the coordinates clicked by the user or the requests that should be sent to the server.

Concretely, the code in Figure 2.3 defines the behavior entry that combines the contents of name and message input fields, wrapping their values in a JavaScript object using the Entry method. The event stream clicks contains an event for every user click on the send button. An event stream of requests to be sent to the server
is then constructed by sampling the behavior entry whenever an event appears on stream clicks and constructing a request from its contents. An unspecified function doPOST ensures that these events are sent to the server (when they appear) using XMLHttpRequests.

Figure 2.3 also defines the event stream chat. It models a network connection on which the client listens to new messages from the server. These messages will contain the updated contents of the chat log and arrive whenever new messages appear on the server. From this stream of chat log updates, an event stream view is constructed that contains updated HTML renderings of the chat log. The render function ensures that these updates will be applied in the displayed web page.

Note that functions like asEventStream, render and doPOST link the FRP application to the outside world. In a pure FRP setting, they (or the primitives they make use of) should be considered as APIs that are part of the framework, not the application.

To make this client operational it needs an accompanying server. For the sake of brevity we will not include the server code, but merely sketch it. Besides serving the client web page and the code in Figure 2.3 to the client, the server needs to listen to client requests on example.com/input. Received messages from clients are added to the chat log which is kept around as the server’s state. When the chat log is updated, the server will push new messages to all clients listening on example.com/output. The book-keeping of the client connections listening on example.com/output can typically be taken care of by a library or framework.

In a typical implementation, the server would be implemented using imperative callbacks. Concretely, there would be a request handler responding to incoming requests on example.com/input, imperatively updating the server state and sending out messages to clients on example.com/output. A less common alternative is to write the server-side part as an FRP application as well, using server-side FRP frameworks like the Reactive Extensions by Meijer [2010].

Typically, the server-side part of the application would be written in a server-side programming language of choice, resulting in a codebase split between JavaScript and that other language. Alternatives are to use JavaScript on the server as well or using a multi-tier language that allows the client-side code to be written in the same language as the server-side code. Multi-tier languages have the advantage that they often offer features that are not available in JavaScript, such as a type system, class-based object-orientation and more standard semantics. We think the example so far already demonstrates some essential aspects of the FRP approach. If we ignore the interfaces to the outside world, then the code is constructed by composing event streams and behaviors, not by defining imperative callbacks and attaching them to events. Figure 2.4 schematically represents our implementation and at least on the
client side, it approaches the mental model in Figure 2.2 that we started from.

Nevertheless, there are still quite a few opportunities for improving this implementation. We see the following remaining problems:

**Interrupted Reasoning** The schema in Figure 2.4 shows a much more strict separation between the client-side and server-side tier than was present in the mental model in Figure 2.2. Very often, the codebase splits up the application in separate codebases, but even when this is avoided in a multi-tier language, there typically remains a semantics gap between the tiers caused by using FRP on the client side and imperative callbacks on the server side. Even if FRP should be used on both the server and client side (which we have not found examples of), current approaches would still treat both parts as separate applications with separate semantics.

**Duplicate Code** In non-multi-tier languages, our application would typically suffer from significant code duplication. For example, data definitions would need to appear in both the server- and client-side codebase.

**Glue-code** Although this is sometimes taken care of by languages and frameworks, connecting both tiers of the application can involve quite a bit of glue code. In our chat application, the example.com/[in/out]put handlers are good examples. They are boilerplate sections of code that are there for (admittedly important) technical reasons; they are not present in the mental model in Figure 2.2 and we prefer a framework or library to take care of this part of the work completely.
Bootstrap  A problem that we have underemphasized so far is how the client’s view of the chat log is initially populated. In our implementation, this could be handled by making the client issue a request to the server after startup to load the data. However, it is more efficient to distribute the initial data as part of the JavaScript code. However, both approaches require special programmer effort. The first approach requires an additional server-side request handler that can send the chat log contents to clients and additional client-side initialization code and the second approach requires server-side code that injects the current chat log state into the code to be sent to a new client.

We think these aspects of the implementation can be improved by writing the application using our multi-tier FRP approach. We demonstrate and explain this in the next section.

2.2.2 Alternative: Multi-tier FRP

In Figure 2.5, we demonstrate multi-tier FRP for the web with an alternative implementation of our example chat application. The code implements both the client and server part of the application. It is implemented in Scala, using our framework that we discuss in Section 2.5. Let us take a closer look.

A first thing that the reader may notice is that the code sometimes mentions types of the form Rep[T], where T is a normal Scala type. These types come from the use of the JS-Scala library [Kossakowski et al., 2012]. We will introduce JS-Scala in Section 2.3.2 and our use of it in Section 2.4.2. For now, it suffices to know that a value of type Rep[T] represents a JavaScript term of type T.

Other types in the example that deserve some explanation are those of the form ClientBehavior[T], ClientEvent[T], ServerBehavior[T] and ServerEvent[T]. These types correspond to the FRP behaviors and event streams that we encountered in the previous example, except that we now make the distinction between those on the server and client.

The code constructs HTML text input fields name and msg and submit button send. The event stream submit models the user’s submitted messages; it is constructed by combining the behaviors nameV and msgV (representing the contents of the corresponding text inputs) by wrapping their values in the Entry case class and sampling the resulting behavior whenever the clicks event stream (representing button clicks) fires.

\(^2\)We shorten ClientBehavior to Behavior, ClientEvent to Event, ServerBehavior to Behavior and ServerEvent to Event for improving code presentation.
case class Entry(name: String, msg: String)
extends Adt
val EntryRep: (Rep[String], Rep[String]) ⇒ Rep[Entry] =
adt[Entry]

lazy val name: Rep[Input] = text("Name")
lazy val msg: Rep[Input] = text("Message")
lazy val send: Rep[Button] = button("Send")

lazy val submit: Event[Entry] = {
val nameV: Behavior[String] = name.values
val msgV: Behavior[String] = msg.values
val entry = nameV.combine(msgV) { EntryRep(_, _) }

val clicks: Event[MEvent] = send.toStream(Click)
entry.sampledBy(clicks)
}
lazy val serverSubmit: Event[Entry] = submit.toServerAnon

lazy val chat: Behavior[List[Entry]] =
serverSubmit.fold(List.empty[Entry]) { (acc , entry) ⇒
entry :: acc
}

def template(view: Rep[List[Entry]]): Rep[Element] =
chat.toAllClients.map(template)

implicit val itemFormat = jsonFormat2(Entry)

Figure 2.5: Example chat application implemented using multi-tier FRP for the web.

In the next step, the event stream submit is sent to the server side, using the toServerAnon method. Note how this method transforms a ClientEvent[Entry] to a ServerEvent[Entry], as expected. The toServerAnon is one of the simplest communication primitives in our API. It is anonymous in the sense that the server cannot determine from what client an event on the resulting stream originated (see Section 2.4.4 for alternative APIs when this is not sufficient).

From the serverSubmit event stream, the chat behavior is constructed, containing the accumulated server state as a list of all entries in the chat log. It is constructed using the fold FRP primitive which takes an event stream to accumulate over, an initial value and a method that adds a new event to the accumulated state, similarly to well-known fold or reduce methods for accumulating over collections.

The accumulated server state is sent back to the client using the toAllClients method. The result is used to fill a template of the web page. We have omitted the template implementation for brevity, but note that it must include the text inputs and
submit button constructed previously, for everything to work.

Finally, the code in Figure 2.5 contains some technical definitions that are needed to make everything work. We use a standard Scala case class Entry, but we need some technical tools to work with it. Specifically, we use the JS-Scala adt macro (that we do not go into details for) to build EntryRep, a function to construct values of type Entry in JS-Scala code. Secondly, we use spray’s jsonFormat2 to automatically construct itemFormat: a JSON serializer and deserializer for our Entry type. The value is required implicitly for the calls to toServerAnon and toAllClients.

In summary, the implementation in Figure 2.5 shows the essence of our multi-tier FRP for the web approach. Both server and client parts of the chat application are implemented as a single FRP application, with special primitives toAllClients and toServerAnon used for explicitly crossing the tier boundary. If we take another look at the problems discussed in the previous section, we can see that our approach provides several improvements.

**Duplication and Glue-code**  Our use of a multi-tier framework, as well as the use of a single FRP network covering both tiers of the application, solves the problems related to interrupted reasoning, code duplication and glue code that we discussed before. Like in other multi-tier frameworks, there is no more duplication of data type definitions and the application is not split into several independent parts.

**Interrupted Reasoning**  Unlike existing multi-tier languages and frameworks, our use of a single FRP network that covers both tiers, ensures that we benefit from the advantages of Functional Reactive Programming on both tiers and that there is a smaller semantic gap in the treatment of asynchronicity on both tiers. Glue code between client and server is removed by our use of toAllClients and toServerAnon communication primitives.

**Bootstrap**  Interestingly, our use of the toAllClients communication primitive, which sends a server behavior to the client allows us to efficiently solve the bootstrapping problem as well. Let us explain how this works. Consider the chat server-side behavior (modeling the server-side state of the chat log) that is sent to the client using the toAllClients method. When a client connects to the server, it needs to know the initial values of the behavior, in order to calculate the initial value of its other behaviors, specifically the ones needed for constructing the initial version of the web page like our chat.toAllClients. Our framework automatically takes care of this, by including the most recent value of the chat behavior in the code that is sent to the server. This is very efficient since no further network requests to the
server are needed before the initial display of the web page. However, it is completely taken care of by the framework without requiring special programmer effort.

Note that we are not saying that our approach is the only one solving some of the above problems. However, we do believe that our approach is unique in solving all of them. Specifically, we use FRP on both tiers for modeling asynchronous code and there is no gap (in terms of programming languages or semantics) between the implementations on both tiers. Furthermore, we benefit from standard advantages of multi-tier languages or frameworks, specifically the fact that we avoid duplicating code, we use a single programming language and we can exploit the static type system and other features of a powerful programming language like Scala.

With this informal introduction, we hope the general idea of multi-tier FRP for the web is clear. In the next sections, we make the idea more concrete and prove feasibility and usefulness of the approach by showing a possible API design and implementation for the approach in the programming language Scala, using an embedded JavaScript DSL that makes Scala usable as a multi-tier language.

2.3 Background

Before we can explain our proposed API and implementation in the next sections, we need to briefly present some background regarding FRP in general and about the JS-Scala library.

2.3.1 Functional Reactive Programming

FRP is a functional approach to programming with time-dependent values and asynchronous events. We adhere to the primary two concepts that were part of the original development in Fran [Elliott and Hudak, 1997]:

Event Stream An event stream is a channel on which events arrive as values of a given type. Mouse click events, for example, can be represented by values containing meta-data regarding the click (e.g. position, button pressed). An event stream Event[T] can be thought of as a continuously expanding list of type List[(Time, T)].

Behavior A Behavior[T] is an abstraction representing time varying values of type T. It can semantically be thought of as a function of type Time → T; a function that always returns a value which may or may not depend on time. The user cursor position, for example, is a behavior since it is always defined and may vary over time.
FRP is a way of reifying event streams and time varying values into first class citizens so interactive applications can be modeled in a declarative manner by combining behaviors and events into an actual application. For code examples, we refer to the code shown in Section 2.2.

It is not our intention to give a thorough overview here, but we mention that FRP comes in many variations. For the purposes of this chapter, we use a discrete time model (i.e., the values of behaviors change discretely, not continuously).

2.3.2 JS-Scala: writing JavaScript in Scala

For representing JavaScript code in the client side of our API, we use JS-Scala: an Embedded Domain Specific Language (EDSL) for writing JavaScript programs within Scala in a natural syntax. The library builds on LMS by Rompf and Odersky [2010]: a framework for embedding DSLs inside Scala, supporting modular DSLs and DSL interpretations.

In JS-Scala, a JavaScript block that produces a value of type \( T \) is represented by a Scala value of type \( \text{Rep}\[T\]}. The following example is a JavaScript program built with JS-Scala (note its type):

```scala
def main(): Rep[Unit] = {
    val name: Rep[String] = prompt("What's your name?")
    println("Hello, " + name + "!")
}
```

The JS-Scala code generator can be used convert this code to JavaScript, resulting in the following output:

```javascript
function main() {
    var x0 = prompt("What's your name?");
    var x1 = "Hello, " + x0;
    var x2 = x1 + "!";
    var x3 = console.log(x2);
}
```

For technical reasons related to JS-Scala’s reification of side-effecting statements in the JavaScript DSL, code that contains JS-Scala scripts will sometimes be constructed as lazy values in our code samples. This ensures that their effects are properly reified as part of the correct JavaScript block.

---

3) JS-Scala should not be confused with Scala.js [Doeraene, 2013] which is a Scala to JavaScript compiler.
2.4 A Scala API for multi-tier FRP

The Scala API that we propose consists of different parts that we will explain in the following sections. We start with the server-side FRP APIs, which are quite standard. Next, we present the client-side API which is similar except that it needs to be adapted to the use of JS-Scala to support generating JavaScript code. Finally, we present the communication primitives that make the link between both tiers. The entire API is written in Scala’s object-oriented style.

2.4.1 FRP API on the server

The two main classes for the Server API are Event\textsubscript{S}[T] and Behavior\textsubscript{S}[T]. Figure 2.6 lists the available methods using the notation

\texttt{instance\_tier\_method[type param](args): return type}

Let us briefly introduce the individual methods.

- \texttt{map} The map transformation applies a given function to every occurrence or update of an event stream or behavior to create a new result.

- \texttt{merge} Interleaves the occurrences of two event streams. For example, two button event streams can be merged into a new one that contains occurrences for both buttons. If both event streams fire at the same time, the first event will precede the second.

- \texttt{filter} Filters events on an event stream according to a given predicate.

Figure 2.6: Server FRP API

\begin{verbatim}
Event\textsubscript{S}[T].map[A](f: T \Rightarrow A): Event\textsubscript{S}[A]
Event\textsubscript{S}[T].merge[A >: T](e: Event\textsubscript{S}[A]): Event\textsubscript{S}[A]
Event\textsubscript{S}[T].filter(p: T \Rightarrow Boolean): Event\textsubscript{S}[T]
Event\textsubscript{S}[T].hold[U >: T](i: U): Behavior\textsubscript{S}[U]
Event\textsubscript{S}[T].fold[A](i: A)(f: (A, T) \Rightarrow A): Behavior\textsubscript{S}[A]

Behavior\textsubscript{S}[T].map[A](f: T \Rightarrow A): Behavior\textsubscript{S}[A]
Behavior\textsubscript{S}[T].sampledBy(e: Event\textsubscript{S}[_]): Event\textsubscript{S}[T]
Behavior\textsubscript{S}[T].fold[A](i: A)(f: (A, T) \Rightarrow A): Behavior\textsubscript{S}[A]
Behavior\textsubscript{S}[T].combine[A, B](b: Behavior\textsubscript{S}[A])
  (f: (T, A) \Rightarrow B): Behavior\textsubscript{S}[B]
\end{verbatim}
hold  Transforms an event into a behavior by holding the latest event value that fired. When no event has fired yet, the resulting behavior has the given initial value.

fold  From an event, an initial value $i$ and a stepper function $f$, $fold$ produces the behavior of accumulated values.

sampledBy  Takes a snapshot of a behavior whenever a given event fires.

combine  Composes the values of two behaviors using a provided combination function. For brevity we only list one combine method, our actual API contains combine methods for an arbitrary amount of behaviors.

### 2.4.2 FRP API for the Client

The client part of our proposed Scala API is shown in Figure 2.7. With $Event_c$ and $Behavior_c$ replacing $Event_s$ and $Behavior_s$, it is similar to the server API except for one aspect. The difference is that most of the operations require parameters of the staged form $Rep[T]$ instead of $T$. Compare, for example, the method signatures for Events’ $map$ method on both tiers:

$$Event_c[T].map[A](f: Rep[T] ⇒ Rep[A]): Event_c[A]$$

$$Event_s[T].map[A](f: T ⇒ A): Event_s[A]$$

The $map$ method for an $Event_c[T]$ takes a function of type $Rep[T] ⇒ Rep[A]$ instead of $T ⇒ A$. This ensures that the function passed to the client-side map is a JS-Scala function that represents and can be translated to JavaScript code. Other client-side APIs use $Rep[_]$ types for the same reason.
2.4.3 Client-Server Interaction

In order to cross the boundary between the two tiers, we provide special communication primitives. Our primitives are methods for events and behaviors that allow sending them from client to server or vice versa. In the background, the framework will implement this by sending requests between the tiers when events arrive as appropriate. The simplest of the methods that we offer are the following:

\[
\text{Event}^C[T: \text{JSJsonW}, \text{JsonR}].\text{toServerAnon}: \text{Event}^S[T]
\]

\[
\text{Event}^S[T: \text{JsonW}, \text{JSJsonR}].\text{toAllClients}: \text{Event}^C[T]
\]

The notation \( T : \text{JSJsonW} : \text{JsonR} \) indicates a context bound and can be used as a Scala analogue to type classes [Oliveira et al., 2010]. We use four type classes \( \text{JsonR}, \text{JsonW}, \text{JSJsonR} \) and \( \text{JSJsonW} \) to require the availability of a function for reading or writing a JSON encoding for a type \( T \) in either server or client code. The client-server interaction methods can only be used for types \( T \) for which the appropriate type class instances are available.

We also offer the following API extension for behaviors:

\[
\text{Behavior}^S[T: \text{JsonW}, \text{JSJsonR}].\text{toAllClients}: \text{Behavior}^C[T]
\]

We believe sending a behavior from a server to clients can be given a precise semantics. For practical purposes, it is in fact essential in our multi-tier setting for allowing clients access to the latest server-side state when they start. The chat behavior in Figure 2.5 shows that tier-crossing primitives are quite natural and practical in a multi-tier FRP web application. In Section 2.5.2, we explain how the primitive can be implemented efficiently, by pre-loading client code with the latest values of the behaviors that are sent to the client.

Finally, we point out that we do not offer an API for sending a client-side behavior to the server. More about this in Section 2.4.4.

2.4.4 Distinguishing Clients in the API

The reader may have noticed that until now, our API makes an important simplification: clients cannot be distinguished on the server. The previous example sends events from the client anonymously (i.e. no way to learn what client an event came from) and server update as a broadcast (i.e. no way to limit the clients that receive an event or send different values to different clients). With such a limitation it would be impossible to write genuine web applications, which send different data to different clients and treat data coming from them differently. For a concrete example, consider how a chat application might send each client only his own private messages or only allow them to see a chat log after they are authenticated).
We therefore propose the following additional communication primitives (type requirements omitted for brevity):

\[
\text{Event}^C[T].\text{toServer}: \text{Event}^S[\text{Client} \Rightarrow \text{T}]
\]
\[
\text{Event}^S[\text{Client} \Rightarrow \text{Option}[T]].\text{toClient}: \text{Event}^C[T]
\]
\[
\text{Behavior}^S[\text{Client} \Rightarrow \text{T}].\text{toClient}: \text{Behavior}^C[T]
\]

We chose to add client information explicitly to the results, effectively providing access to information that was lost using the APIs presented earlier.

As we will explain, these primitives are more general than the simpler ones in the previous section, i.e. the old ones can be implemented in terms of the corresponding new one.

\text{Event}_c.\text{toServer} \quad \text{The first method should be read with the intuition that a client event contains more information than just the event occurrence and value. Since it is bound to one client it also contains the client's identity. To allow full access to the information of a client event on the server we propose the API above. The Client value in the returned server event is an opaque but comparable value, which uniquely identifies a client.}

\text{Event}_s.\text{toClient} \quad \text{The second method on the other hand, involves transforming event occurrences on the server into occurrences on the client. This primitive allows the programmer to define per client whether it should receive an event and if so, what value should be sent. More concretely, when the server event stream fires with an event value of type Client \Rightarrow \text{Option}[T], then only the clients for which this function returns \text{Some} x will receive the event and they will receive x as the event value.}

\text{Beh}_a.\text{toClient} \quad \text{Finally, when sending a behavior from the server to the client, we also want to allow showing different values to different clients. Concretely, when the server behavior contains a value of type Client \Rightarrow \text{T}, the sent behavior will contain, on every client, the function's result for that client.}

\text{Beh}_c.\text{toServer} \quad \text{We previously mentioned that we did not implement tier-crossing primitives for client behaviors to the server. Given that clients' lifetimes in a web setting are subintervals of the server's, a plausible design would be to provide an API primitive of type:}

\[
\text{Behavior}^C[T] \Rightarrow \text{Behavior}^S[\text{Map}[\text{Client}, T]]
\]
This primitive would produce a server behavior that represents the value of the behavior in each individual client. However, we found the type signature rather clunky and the added value limited, so we decided not to include it.

To demonstrate the power and flexibility of the extended API, Figure 2.9 shows an example application that extends our previous chat application. This time we create a chat application with support for private messaging with a UI that looks something like Figure 2.8. The listing shows the full code with just the UI template omitted for brevity.

## 2.5 Implementation

We have implemented the proposed APIs in a proof-of-concept multi-tier FRP web framework in Scala. The implementation can be downloaded from Github[^github-repo], together with instructions for building the code and experimenting with it.

Our implementation makes use of a composable web server abstraction called a *route*, that (to our knowledge) was made popular by Sinatra [Mizerany, 2007]. For our purposes you may think of a route as a combination of an URL and its corresponding functionality. The implementation of routes that we are using is the Spray library [Rudolph and Doenitz, 2012].

### 2.5.1 Client & Server FRP API

To implement our API with minimal effort we rely on an existing JavaScript FRP framework called BaconJS [Paananen, 2012]. Our Event\(_c\) and Beh\(_c\) are implemented[^github-repo]

[^github-repo]: https://github.com/Tzbob/s-mt-frp/releases/tag/onward14
case class Entry(name: String,  
    target: Option[String],  
    content: String) extends Adt
val EntryRep = adt[Entry]
implicit val itemFormat = jsonFormat3(Entry)

case class Chat(pub: List[Entry] = Nil,  
    priv: Map[Client, List[Entry]] = Map() withDefaultValue Nil)

case class View(pub: List[Entry], priv: List[Entry])  
    extends Adt

implicit val viewFormat = jsonFormat2(View)

lazy val name: Rep[Input] = text("Name")
lazy val msg: Rep[Input] = text("Message")
lazy val target: Rep[Input] = text("Target")
lazy val send: Rep[Button] = button("Send")

lazy val submit: Event[Entry] = {
  val combined: Behavior[Entry] = 
    name.values.combine(target.values, msg.values) {
      (n, t, m) ⇒
      EntryRep(n, if (t == "") none else some(t), m) } 
  combined.sampledBy(send.toStream(Click))
}

lazy val onServer: Event[(Client, Entry)] = submit.toServer

lazy val chat: Behavior[Chat] = 
  onServer.fold((Map[String, Client](), Chat())) {
    case ((ppl, Chat(pub, priv)), (sender, entry)) ⇒
      val newPpl = ppl + (entry.name -> sender)
      val newChat = entry.target match {
        case Some(t) ⇒
          def cons(c: Client) = c -> (entry :: priv(c))
          c.copy(priv = priv + cons(ppl(t)) + cons(sender))
        case None ⇒
          c.copy(pub = entry :: pub)
      }
    (newPpl, newChat)
  }.map { case (map, chatLog) ⇒ chatLog }

lazy val chatVw: Behavior[Client ⇒ View] = 
  chat.map {
    case Chat(pub, priv) ⇒
      client: Client ⇒ View(pub, priv(client))
  }

def template(view: Rep[View]): Rep[Element]
def main: Behavior[Element] = chatVw.toClient.map(template)

Figure 2.9: Multi-tier FRP chat application with private messaging
using BaconJS representatives underneath. The implementation consists essentially of a BaconJS event or behavior (defined in JS-Scala) and an optional route. The optional route comes into play for implementing the communication primitives (see below).

\[ \text{Event}_c[T] \approx \text{Option[Route]} \times \text{Rep[BaconEvent}[T]] \]

The methods from the standard FRP API on Event\(_c\) and Beh\(_c\) are simply delegated to the BaconJS representative to create a new Event\(_c\) containing the untouched previous route and the new BaconJS result.

The implementation of Event\(_s\) and Beh\(_s\) is very similar. On the server, we rely on a Scala FRP implementation named Scala-reactive [Gugenheim, 2011]. As for client-side event streams and behaviors, the implementation is a pair between an optional route and its Scala-reactive counterpart:

\[ \text{Event}_s[T] \approx \text{Option[Route]} \times \text{ReactiveEvent}[T] \]

Operations on server-side events and behaviors are delegated to their counterparts as well.

### 2.5.2 Basic Client & Server Interaction

Implementing the communication primitives from Section 2.4.3 requires generating appropriate glue code. Although our APIs keep the client-server boundary visible to the programmer and he/she still decides when it is crossed, the technical machinery for actually sending events and behaviors is abstracted away. In this section, we briefly discuss how the different primitives are implemented, using sections of pseudo-code to convey the core idea.

**Event\(_c\).toServerAnon**  
For sending a client-side event stream to the server, the first thing that we create is a fresh URL on which the server and client will communicate:

```scala
val url = URLEncoder.encode(UUID.randomUUID.toString, "UTF-8")
```

On the client, we generate an imperative callback that pushes events towards the server when they arrive. This is done by issuing POST requests to the generated URL.

```scala
def init\(_c\)[T: JSJsonW](e: Event\(_c\)[T], url: String) =
  e.baconRep.onValue(fun { value ⇒
    doPost(url, value.toJSONString) })
```

The callback is attached to the BaconJS primitive underneath the target Event\(_c\) so that it is executed when a new event value arrives. Note that this client initialization
code is not explicitly kept track of as part of the returned server event stream. Instead, by issuing the commands during the initialization of the application, we make use of JS-Scala’s reification of effects to include the code in the generated JavaScript client initialization code.

The server-side glue code that is needed for implementing the \texttt{Eventc.toServer} primitive is a handler for the aforementioned POST requests. We create a Scala-reactive EventBus, essentially an Event on which we can imperatively push data. Every time a new request comes in, the following route implementation will decode the contained JSON data using the appropriate \texttt{JsonR} method and push the decoded value onto the corresponding EventBus.

\begin{verbatim}
def mkRoute[T: JsonR](url: String, tgt: EventS[T]) =
  path(url) { post {
    entity(as[String]) { data ⇒ complete {
      tgt.reactiveRep.push(data.asJson.convertTo[T])
    }}
  }}
\end{verbatim}

The \texttt{Eventc} that we return will contain the above route, composed with the routes of the original \texttt{Eventc}. It will have the created EventBus as the underlying event stream.

\texttt{Eventc.toAllClients} Sending a server-side event stream to the client follows a similar process. Our implementation uses Server-Sent Events (SSE, part of HTML5) but Websockets could be used instead. SSE allow clients to connect to listening URLs and keep the connection open for the server to push chunks of data.

Again, we start by generating a fresh URL for communicating over. A route is created for this URL and whenever a client connects to it, we connect a callback to the Scala-reactive event stream underlying the \texttt{Eventc}. This callback will push events to the listening client.

\begin{verbatim}
def mkRoute[T: JsonW](url: String, evt: EventS[T]) =
  path(url) { get {
    respondWithMediaType('text/event -stream ') {
      startHTTPChunk(ctx)
      evt.reactiveRep.foreach { data ⇒
        sendChunk(ctx , Chunk(s"data:" + data.toJson + "\n\n"))
      }
    }
  }}
\end{verbatim}

This route for the listening URL will be included in the \texttt{Eventc} returned from the \texttt{Eventc.toAllClients} call.

On the client side we create a Bacon EventBus, which is similar to Scala-reactive’s. The \texttt{Eventc} that we will return will have this EventBus as its underlying BaconJS event stream. We use a client-side EventSource (the standard client-side interface to
listen to Server-Sent Events) to connect to the generated URL. The received messages are converted from JSON and injected into the EventBus.

```scala
def init[T: JSJsonR](evt: Event[T], url: String) =
  EventSource(url).onmessage =
    fun { ev: Rep[DataLiteral] ⇒
      evt.baconRep.push(ev.data.toJson.convertTo[T]) }
```

Behₙ.toAllClients Sending server behaviors to client behaviors requires additional work. We can consider a behavior as an initial value together with an event stream of updates. Sending the changes is easily implemented in terms of the previously presented Eventₙ.toAllClients method. We will now explain how we transfer the initial value to the client, so that it can be combined with the event stream of changes to form the full behavior.

As mentioned before, our idea is to include this initial value in the JavaScript code that is sent to the client when it first connects. The behavior that we return from Behₙ.toAllClients, is constructed using the following code:

```scala
def json(): Rep[String] =
  unit(serverBehavior.reactiveRep.now.toJson)
val changes = serverBehavior.changes.toAllClients
changes.hold(delay(json).asJson.convertTo[T])
```

We use \(\text{unit}(t : \text{String}) : \text{Rep}[\text{String}]\) to lift a constant Scala value into the JS-Scala program. The initial value is constructed as the \(\text{json}\) thunk above (a function of type () ⇒ Rep[String]). It is wrapped in a special \(\text{delay}\) call that will cause it to be evaluated only when the client code is being prepared for sending to a new client. At that moment, the ‘current’ value of the server behavior will be converted to JSON and included in the code.

### 2.5.3 Client Aware Interaction

Implementing the client aware APIs from Section 2.4.4 requires some extensions to the basic implementation.

Eventₙ.toServer We modify the preceding implementation of toServerAnon to send an Eventₙ[T] to an Eventₙ[(Client, T)] by generating a unique identifier for a client when it first connects and injecting this identifier into the code generated for that client.

The client pseudo-code for Eventₙ.toServer is then extended by passing the client identifier as a parameter in the target URL.
def initClient[T: JsonW](e: EventBus[T], url: String) =
  e.baconRep.onValue(fun { value ⇒
    doPost(delayForClientId { id ⇒
      url + "/?id=" + id }, value.toJSONString)
  })

delayForClientId is an extension of the previously mentioned delay method. Instead of thunks of type () ⇒ Rep[T], it allows including thunks of type Client ⇒ Rep[T] in client-side code, so that the correct client identifier is filled in when generating JavaScript code for that client.

The server-side code for EventBus.toServer now extracts the client identifier from the request, uses it to construct a correct client object and includes it in the produced event value.

def mkRoute[T: JsonR](url: String ,
  tgt: EventBus[(Client , T)]) =
  path(url) { parameter('id) { id ⇒
    post {
      entity(as[String]) { data ⇒ complete {
        tgt.reactiveRep
          .push((Client(id), data.asJson.convertTo[T]))
      }}
    }
  }}

EventBus.toClient  The new EventBus.toClient will be invoked on event streams of type Client ⇒ Option[T]. The server-side glue code is changed to evaluate the event values (which are now functions!) for the identifier of the client that connects to the listening URL and (optionally) push the result information to the client.

def mkRoute[T: JsonW](url: String ,
  evt: EventBus[Client ⇒ Option[T]]) =
  path(url) { get { parameter('id) { id ⇒
    val client = Client(id)
    respondWithMediaType('text/event-stream') {
      startHTTPChunk(ctx)
      evt.reactiveRep.foreach { fun ⇒
        fun(client).foreach { data
          sendChunk(ctx,
            Chunk(s"data:" + data.toJson + "\n\n"))
        }
      }
    }
  }}

The changes required for the discussed implementation of Behav.toAllClients are similar to the ones just discussed, so we omit them for brevity.
2.6 Conclusion

In this chapter, we propose multi-tier FRP for the web, an approach for developing reactive web applications in an elegant way by combining both FRP and multi-tier languages. The approach simultaneously solves existing problems such as (1) interrupted reasoning in multi-tier web development, (2) the complex control flow when using imperative callbacks to handle asynchronous user and network I/O and (3) the problem of bootstrapping clients.

FRP reifies time dependent data such as event streams and time-varying values, allowing a programmer to model his time dependent problems rather than handle them. Our proposal of spanning the FRP paradigm across client- and server-side tiers, allows writing a single FRP program whose semantics spans both tiers rather than two programs whose separate semantics combine to a full web application, thus removing a semantic gap within an application. Practically, our approach has the potential to reduce the amount of technically important yet difficult and repetitive boilerplate code in applications. Additionally, the client bootstrapping problem is handled elegantly and implicitly through the primitive for sending server behaviors to the client.

We have made our multi-tier FRP for the web more concrete with a proposal for a Scala API, as well as an implementation built on existing technology like JS-Scala, BaconJS, Spray and Scala-reactive. The API and implementation allow us to demonstrate the potential of the general approach. They also show some possible solutions for some of the problems that arise, such as exposing client identity on the server and producing separate client and server code from the single code base of a multi-tier FRP application.

2.7 Future Work

We believe the results in this chapter show the feasibility and the potential of multi-tier FRP for the web as a new approach to the declarative development of interactive web applications. However, quite some interesting questions remain to be solved before this can become fully practical. We see the following interesting directions for future work. Most of the tracks discussed here should be tackled at the level of the general approach as well as that of our Scala API design and implementation.

Continuous Time Model We spent little time in this chapter explaining the differences between continuous or discrete time FRP and the impact of that choice on our design. First, note that some of our primitives (e.g., Beha.fold) are only
meaningful in the discrete time model that we use. Secondly, it is important to understand that the choice between a continuous and discrete time model has implications for the choice between a pull- or push-based evaluation strategy. We use a discrete time model and thus a push based strategy to allow efficient tiered updates. If we had used a continuous time model, we would have needed to use a pull-based evaluation in our FRP network, including the client-server primitives (i.e., using a primitive like websockets instead of XMLHTTPRequests, for example). For future work we would like to further research the implications of continuous time models and pull-based evaluation on multi-tier FRP in general. Also Elliott [2009]'s work on combining push and pull evaluation semantics seems a promising addition to multi-tier FRP, as it might allow more fine-tuned access to resources like the DOM or databases.

In Chapters 4, 5 and 6, we add a behavior that has pull-based evaluation. tier-crossing primitives are available on push-based behaviors (discrete behaviors) while pull-based behaviors are used to model state that lives outside of the FRP program such as the DOM.

**Atomicity** In FRP, when a primitive behavior changes value or a primitive event stream fires, the semantics dictate that all behaviors and events that depend on it should correctly reflect the changes before their new values are made visible to the outside world (e.g. by updating the web page visible to the user). This requires calculating a correct dependency order for the FRP network. The term *glitches* is used to describe situations where an FRP implementation makes new values visible that do not yet correctly reflect the changed values. The term glitch is used because it is often a temporary problem and the correct value is propagated quickly afterwards. Such glitches may result in inconsistencies in the application state and many FRP frameworks work hard to prevent them.

Within both the client- and server-side tier, we do not expect any problems related to glitches. Our implementation relies on the underlying frameworks (BaconJS and Scala-reactive) to properly ensure correct updates to the network.

When it comes to our communication primitives, we intend these to have a semantics similar to Elm’s `async` [Czaplicki and Chong, 2013]. This means that every event on the original stream will be propagated to the FRP network on the other end and will be processed separately from other events.

Although the semantic correspondence to Elm’s `async` strengthens our trust in the described semantics, we still plan to experiment with an alternative semantics for the communication primitives in future work. Under this alternative semantics, all changes or events that should be propagated to the other tier would be collected during a propagation cycle. They would then be shipped to the other tier as part
of a single network message and be processed in a single time instant on the other tier. We have the intuition that this would make it more practical for programmers to avoid glitches. It might also fit better into a denotational semantics for the FRP framework, but this remains to be confirmed.

This implementation is discussed in detail in Chapter 6 where we discuss our tiered glitch-freedom that is also reflected in the denotational semantics in Appendix A.

**Error Handling** We have neglected proper error handling in our proposal, the API shown in this chapter has no way of handling communication problems between server and client. An approach we have been thinking about is another extension of the interaction methods to incorporate a behavior that represents the current status (Connected, Pending, Disconnected) of the accompanying connection.

\[
\begin{align*}
\text{Event}^C[T].\text{toServer}: & (\text{Event}^S[(\text{Client}, T)], \text{Behavior}^C[\text{Status}_c]) \\
\text{Event}^S[\text{Client} \Rightarrow T].\text{toClient}: & (\text{Event}^S[T], \text{Behavior}^S[\text{Status}_s])
\end{align*}
\]

Another alternative may be to provide just two primitives `networkProblem_c` and `networkProblem_s`, of types `Event_c[ProbDesc]` and `Event_s[ProbDesc]` that can be used for signaling problems of any connection.

**Performance** In our chat application example, we model the entire application state as a behavior on the server that we make available on the client side by bringing it to the client tier. This has an impact on performance due to the nature of behavior updates. They are updated by replacing the current value with a newer value: \(\text{Beh}[T] \approx T \times \text{Event}[T]\). In the case of our example this means that updates containing the entire chat log would be sent over the network to the client. It would be more efficient to send just the new chat messages to the client and reconstruct the full chat log on the client side from its previous value and the new messages received.

We want to approach this problem in the future using the concept of incremental behaviors. An incremental behavior is a behavior that can be considered as an initial value together with an event stream of increments: \(\text{IncBeh}[T] \approx T \times \text{Event}[T \Rightarrow T]\). This approach for FRP has been researched before in Scala by Maier and Odersky [2013] and we think it could allow us to encode increments and sending them over the network instead of the entire state.

**Persistence** To support non-trivial web-applications, an FRP persistence API seems essential. We plan to investigate the design of an API and implement it by making use of the notifications of changes that some databases can send. As far as we know, this has not yet been attempted in an FRP API.
Scalability (Replication)  Many modern web-frameworks support replicating the server part of a web application over multiple servers for scalability reasons. Because of the fold-related primitives in our API, our framework is stateful on the server, which makes it impossible to scale out using traditional replication. After defining a persistence API, we intend to investigate a solution for replication based on removing these primitives in the server API and instead only allow folding in the persistence API.

Security  We make no claims for security of our implementation yet. The use of UUIDs and their possible predictability could be a security vulnerability. We only provide a means to distinguish between client connections and provide no methods yet for authentication and authorization.
3  |  Scalagna: towards a Multi-tier Scala

As a basis for the rest of the multi-tier FRP framework we looked into using Scala as a multi-tier language. The previous chapter used JS-Scala, a Scala embedded domain specific language for JavaScript. The work described in this chapter uses Scala.JS instead, a mature and full-fledged Scala to JavaScript compiler with its own ecosystem of libraries. The goal is to use Scala as a multi-tier language without modifying its build system or any of its compilers.

The research described in this chapter contains the following paper:


The research for this chapter was based on my own ideas, the implementation and supporting work was done by master thesis student Michael Greefs. The paper was written by me with feedback and comments from Dominique Devriese and Frank Piessens.

Abstract

In the state-of-practice, developing web applications requires dealing with multiple programming languages or codebases. To address this issue, researchers have proposed multi-tier languages such as Hop or Links that support client and server development in a single language and in one codebase. Even if such multi-tier languages are often strongly based on an existing language - for instance Hop is based on Scheme - they are new languages, and require a new compiler.

The objective of this paper is to define a multi-tier language as a library-based Scala DSL. Scala already supports compilation to both the Java VM and to JavaScript. The
multi-tier language we propose in this paper, Scalagna, combines the existing Scala JVM and JavaScript ecosystems into a single programming model without requiring changes to, or rewrites of the existing Scala compilers. We discuss how this is possible using Scala’s excellent support for defining DSLs, and the experimental Scala macro system. We show that Scalagna has reasonable performance, and by porting an existing Eliom application, we provide evidence that Scalagna is as expressive as other established and existing multi-tier languages.

3.1 Introduction

Developing web applications presents a number of interesting challenges for programmers. One challenge is due to the inherent distributed nature of the platform with parts of the application running on the server and other parts on the (zero or more) clients. This causes an impedance mismatch within applications, for example, keeping client and server programming interfaces synchronized is difficult and tedious. Multi-tier programming languages [see among others Chlipala, 2015, Serrano and Queinnec, 2010, Radanne et al., 2016] aim to resolve this problem by allowing both the client and server parts of a web application to be written in a single language and codebase. These existing multi-tier languages are new programming languages requiring a new compiler, and hence a significant initial development cost.

Our objective in this paper is to investigate whether we can get the benefits of a multi-tier language without the cost of having to develop a new compiler. More specifically, we aim to implement a Domain Specific Language (DSL) for multi-tier programming as a Scala library, without the need for a new compiler (or changes to existing compilers). We start from Scala, because it already has all the key components. Scala is a language with a healthy amount of libraries. Its primary runtime is the JVM and it boasts fluent Java interoperability. In addition, Scala.js is a plugin that compiles Scala to JavaScript with strong focus on interoperability with existing ecosystems Doeraene et al. [2016].

The key contributions of this paper are:

- We design and implement Scalagna, a multi-tier programming environment for Scala based on the existing JavaScript and JVM ecosystems that requires no additional compiler changes or plugins. It uses Scala’s experimental macros to do minor rewrites as explained in section 3.4.

  Scalagna is publicly available at https://github.com/tzbob/scalagna.

- We quantify the performance cost of Scalagna by comparing the performance of a Scalagna application with the same application developed directly in Scala.
• We evaluate the expressiveness and verbosity of Scalagna through a comparison with Eliom, an established multi-tier language extension to OCaml.

3.2 Scalagna by Example

Scalagna web programs are written in a single codebase. Two programs are extracted, one program runs on the web-server while the other runs on the browser. In this section we give you a feel of what it would be like to program with Scalagna before we go into the nitty-gritty of the system. We focus on features and solutions that are needed in a modern client/server web application.

3.2.1 Pages & Annotations

In Scalagna, you can write client and server code within the same codebase. Annotations are used to mark the destination of a code block. For example, if we want to create a page to print two statements, on the client and server respectively:

```scala
object Prints extends MTPage {
  override def execute(): List[MTService] = {
    @server val head = Nil
    @server val body = div(h1("Hello"))
    @server val _ = println("one")
    @client val _ = println("two")
    List(htmlService("/prints", body , head))
  }
}
```

A page is defined as a list of services and in this small example we use the HTML service. An HTML service is defined by a path relative to the web server, a body and the head. Both body and head are server values, they represent the HTML content of the served page, in this case, a simple hello title and an empty head.

The print statements are added as annotated nameless values, depending on the annotation the evaluation of the value is server or client side. In this case, as the server starts up, “one” is printed on the server console. Every time a user accesses the “/prints” page on his browser, “two” is printed in the browser’s console.

3.2.2 Exposing Server State

While the previous example showed us how to execute initial side-effects for both the client and server tiers, it did not hint at how we could deal with typical server state. For example, what does a visitor counter look like in Scalagna?
object Visitors extends MTPage {
    @server var counter: Int = 0
    override def execute(): List[MTService] = {
        @server val head = Nil
        @server def body = {
            counter.value += 1
            div(counter.value.toString)
        }
        List(htmlService("/visitors", body, head))
    }
}

Figure 3.1: Visitor Counter

In Scala, mutable and immutable references are explicitly denoted with the `val` and `var` keyword. In the previous example we annotated immutable values, but Scalagna also supports the client and server annotation of variables as well as methods (def) and type aliases (type).

In Figure 3.1, a mutable variable `counter` represents the visitor count. Access to this mutable server variable is available through `.value` on `counter`. The body of the page displays a string representation of the counter. Note it is now defined as a method instead of a value, and the counter is increased every time the body is evaluated. Since Scalagna HTML services re-evaluate body and head on each serve to a client, a new client updates the value of the counter, and the new value is embedded in the page.

While Figure 3.1 shows how to expose server state to the client through HTML, it does not provide a direct solution to using server values in clients, e.g., having a client-side counter. In Figure 3.2, we demonstrate the injection feature. Server values can be injected into client contexts, injected values are added into a client’s context every time a user accesses a page. The example behaves as the first example we showed above. Every time a user accesses the page the browser’s console will show the current visitor count.

### 3.2.3 Dynamic Client/Server Interaction

So far, interactions between client and server have been static. Client values are injected and HTML values are determined when a user requests a page. Modern web applications require more flexibility than this. It is beneficial to the user experience if client code can interact with the server during its execution, it makes interactive applications such as chats possible without reloading pages. These interactions are first-class values in Scalagna. They are added as services.
object VisitorPrints extends MTPage {
  @server @inject var counter: Int = 0
  override def execute(): List[MTService] = {
    @server val head = Nil
    @server def body = {
      counter.value += 1
      div(h1("Hello"))
    }
    @client val _ = println(counter.inject.toString)
    List(htmlService("/visitorprints", body, head))
  }
}

Figure 3.2: Visitor Counter Prints

object FileRead extends MTPage {
  @server val read = (file: String) ⇒ {
    println("Reading file...")
    Source.fromFile(file).mkString
  }
  val rpcFileRead = rpcService("/read", read)
  override def execute(): List[MTService] = {
    @server val head = Nil
    @server def body = div(h1("Hello"))
    @client val _ = rpcFileRead("~/test").foreach(println)
    List(htmlService("/visitorprints", body, head), rpcFileRead)
  }
}

Figure 3.3: File Read

In Figure 3.3, we show an example that uses such a service. The application ignores all security advice and allows direct access to the file system from the browser by exposing an RPC that can read files. The rpcFileRead service is created from a string that represents the path on which it will be made available and a server function. This server function will be executed when the service is used, its results are serialized and sent back to the client. In Figure 3.3, on line 14, the service is used. It can be called as a regular function, however, its return type is wrapped in a Future, i.e., a value to denote an asynchronous operation. With foreach, the success case of the asynchronous operation is handled and our file contents are printed to console.

With these features combined, we have a minimal but powerful way of expressing

---

1We ignore the error case in this contrived example.
web applications as one application. We discuss more features in the next section and a larger example that uses all of our features when we compare Scalagna to Eliom Radanne et al. [2016] in section 3.5 on evaluation.

3.3 API

The examples did not cover the entire Scalagna API. In this section we briefly look at each feature and its types in detail. We largely follow the API of Eliom, making changes where it decreases added complexity to Scala/Scala.js.

Bootstrapping Scalagna programs are defined through a list of pages, which are defined through a list of services:

```scala
trait MTApp { def mtMain: List[MTPage] }
trait MTPage { def execute: List[MTService] }
```

Client/Server Programmers make use of the `@client` and `@server` annotations to mark which sections of the code belong to which tier. Any code that is not marked with such an annotation belongs to both tiers. Note that shared variables are not synced across tiers, they are duplicated. Changing shared variables’ value does not impact their value on other tiers automatically and their use is not recommended.

Each annotation turns a value into its respective tier value: `@client val x: Int = 1` becomes a value `x` of type `MTClient[Int]`, likewise for the server counterpart. Variables turn into `MTClientVariable`s. The API of these tier values is the following:

```scala
trait MTClient[T] { def value: T }
trait MTClientVariable[T] { def value: T
def value_(t: T): Unit }
```

The `value` accesses the client value. If this method is used in a client context it returns the expected value. If the method is used in a server context instead, it fails with a compile-time error, letting the programmer know that that particular value is not available in that tier. Note the extra method definition on the variable – in Scala this enables the setter syntax so that assignment is possible, e.g., `x.value = 5`. The server values and variables behave the same.

Annotations can also be put on methods and type aliases. An annotation on a method rewrites its return type, e.g.: `@server def x(): T` turns into `def x(): MTServer[T]`. Type aliases are even simpler, they are erased from the opposite tier.
**Injection**  It is possible to make server values available in a client context without paying a large cost. Injecting a server value into a client value is done when a page is loaded. Upon load time the current value of the to-be-injected server value is added to the initial payload. Later, an initialization phase in the client program retrieves these injected values. The need for these kind of injections is high when developing client/server web applications. Server-based “initial” values are required all the time, the history of a chat box, an initial playing field of a game, etc.

To inject a server value it first has to be marked as being injectable through an annotation:

```scala
@server @inject val init: Int = 0
@client def return = println(init.inject)
```

An extra requirement for injections is the presence of a type declaration. This type is used to derive the (de)serialization mechanics\(^2\).

**Fragments**  Fragments is something we have not yet touched upon. A fragment is a special type of client value that has meaning on the server. Its use in Scalagna is limited to binding Scala.js functions to server-rendered HTML. This supports convenient generation of HTML pages that use the tag attribute event binding, e.g., `<button onclick=...>...`

An HTML fragment is created by adding an additional annotation to a client value:

```scala
@client @fragment val onClk = (el, e) ⇒ println(el, e)
```

The only fragments that are supported right now are functions of the following type:

```scala
(el: dom.Element, e: dom.Event) ⇒ Unit
```

They take the bound DOM element and an event as input and return nothing. Scalagna supports using fragments in an existing Scala HTML DSL\(^3\), e.g.:  

```scala
@server val body = button(onclick := onClk)
```

**Services**  All services that are available follow the same usage pattern. Anything needed to define the service itself is required in its creation function and all functionality that it provides is available through the service value itself:

```scala
def htmlService(url: String,
    body: ⇒ MTSolver[Seq[Frag]],
```

\(^2\)Note that no custom serialization code was written for Scalagna, it integrates with typical Scala generic serialization libraries that use a combination of implicits and macros.

\(^3\)https://github.com/lihaoyi/scalatags
head: ⇒ MTServer[Seq[Frag]]): MTServiceHTML

def rpcService[U: Writer : Reader, V: Reader : Writer](
  url: String,
  sFunction: MTServer[U ⇒ V]): MTServiceRPC[U, V]

trait MTServiceRPC[U: Reader: Writer, V: Reader: Writer]{
  def apply(ff: U): Future[V]
}

def webSocketService[T: Writer : Reader](
  sFunction: MTServer[T ⇒ Unit]): MTServiceWebSocket[T]

trait MTServiceWebSocket[T: Reader: Writer] {
  def send(message: T)
  def onmessage(f: T)
}

Regarding HTML, in this DSL HTML tags are of type Frag. The MTServiceHTML that is returned is not used for anything but registering the service.

The MTServerRPC service is created from a path and from a server function to define its behavior. Its value is used to make the RPC call in a client context.

The websocket service requires a server function to define the server-side reaction. Its return value can be used to send values and listen to messages on the client side. Note that our service at the moment does not allow the server to initiate messages, instead it acts as a broadcast channel. All client messages are forwarded to all other clients. However, this is purely due to time constraints and there is no technical limitation stopping us from exposing the full websocket API.

### 3.4 Implementation

Due to the limited space we can only provide a high-level overview of the implementation, both the implementation as well as some examples are available online⁴.

As mentioned during the introduction, one of the key design goals of Scalagna was to lift the Scala.js and Scala ecosystem into a multi-tier programming environment with as little additional complexity as possible.

Scalagna is completely library based. It does not require a compiler plugin or changes to Scala.js. It relies on structuring the program in a specific way and uses macros to increase compatibility with existing ecosystems.

⁴https://github.com/tzbob/scalagna
3.4.1 Structure

A Scalagna application has one main method for a user to fill in, one API to create an HTML service, etc., but the same codebase does vastly different things on the client or the server. In practice, a Scalagna application is one codebase used to derive two applications, a JVM application that runs a standard Scala web server and a Scala.js application. The entire structure is shown in Figure 3.4.

The Scalagna codebase is split into three parts, a shared, JVM and a JS part. This is a typical way of writing cross-platform Scala projects, the shared section of the codebase, as the name implies, contains all code that is shared between client and server. The JVM and JS parts can both use the shared section of the codebase but not each other. The Scalagna library is structured so that the API definitions lives in the shared section and tier-specific implementations of them are in the corresponding JS/JVM sections.

For example, in Figure 3.4, we show two implementations of MTApp. One starts a web server and handles serving HTML pages, replying to RPC actions, etc., the other handles initialization of clients e.g., injected (server ⇒ client) values. The same structure is used to implement pages and services.

The implementations of MTPage and the various services work similarly. The JVM implementations of those APIs register the required functionality with the web server while the JS implementations of those APIs do the appropriate calls when needed, e.g., an RPC service is registered on the JVM while the JS version makes calls to the server functionality.
3.4.2 Annotations

The implementation of the annotations in Scalagna are much less complicated than they seem. They are macro based but they do not do any sophisticated checks, they are simple wrappings on the AST. For example, annotating a value with @client causes the client macro to wrap the value to an internal method call. This method call consists of regular, non-macro code, that wraps the value in a MTClient value.

So far this did not require the power of macros, a client method could have done this. However, reusing the Scala.js ecosystem becomes problematic:

```scala
@client val _ = {
  import org.scalajs.dom
  dom.window.alert()
}
```

In this example, the compilation fails on the server since there is no JVM library to supply the package org.scalajs.dom. This is strictly a Scala.js library. We solve this problem in the server version of the @client macro by erasing the client-annotated code before the Scala compiler tries to compile it. As such, the typechecked version of the codebase on the server no longer has the client-side import. This allows Scalagna to support ecosystem-specific libraries in annotated sections. Instead of the removed code, the macro inserts a dummy MTClient on which .value is not a valid call. This follows the same approach as we explained above, MTClient has two implementations, one for server and client.

We do not allow the nesting of macros in Scalagna and for technical reasons require types on injected server values which we also check in the macros. However, the code for an entire @client macro is 130 lines of code with a large amount being static definitions of ASTs.

3.5 Evaluation

The following microbenchmarks are executed on a Windows 10 laptop using an i7-2720QM CPU with the Chrome browser.

**Server Performance**  The server side of Scalagna makes use of http4s\(^5\), a Scala web server. In our first micro benchmark we test a shopping list application. The server section of this application can be seen in Figure 3.5. There are three services, one serves an HTML page, one clears the internal list and one adds a value to the

\(^5\)http://http4s.org/
type ListItem = (String, String)

@server var shoppingList: List[ListItem] = List.empty

@server val addToShoppingList = (e: ListItem) ⇒
  shoppingList.value = e :: shoppingList.value

@server val clearShoppingList: (Unit ⇒ String) = _ ⇒
  shoppingList.value = List.empty

val rpcShoppingList = rpcService[ListItem, String](
  "/shoppinglist", addToShoppingList)

val rpcClearShoppingList = rpcService[Unit, String](
  "/clearshoppinglist", clearShoppingList)

val htmlPage = htmlService("/shopping", ..., ...)

Figure 3.5: Shopping List

Table 3.1: Server Performance (ms)

<table>
<thead>
<tr>
<th></th>
<th># req</th>
<th>average(ms)</th>
<th>req/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scala</td>
<td>78812</td>
<td>12</td>
<td>263</td>
</tr>
<tr>
<td>Scalagna</td>
<td>77858</td>
<td>12</td>
<td>260</td>
</tr>
</tbody>
</table>

internal list. In our benchmark we simulate 2000 users using Locust⁶. In Table 3.1, we show our results. As expected, they are really close, the small difference can be because of the way requests are matched but the differences are so small it could just be benchmark noise.

Client Performance  We test the client overhead by comparing a Scala.js application to the client side of a Scalagna application. The program that we benchmark is the client side of the shopping list application. We measure the overall code size and the time it takes to execute the main function, that is, perform the initialization of the application.

In Table 3.2, we show the impact Scalagna has on page load times. We use the Google Chrome Devtools and separate our measurements into two pieces, the time it takes to parse and execute the script and the time it takes to execute the main function. For handwritten Scala.js code the initial loading time is higher because the code is handwritten for the problem and no injection is used to get values from the server into Scala.js. Contrary to Scalagna, where injections are used and the initial executing of the script does barely anything.

⁶https://locust.io/
### Practical Comparison with Eliom

As a test of Scalagna’s expressiveness and verbosity we make a direct comparison with Eliom, a multi-tier language based on OCaml. Eliom provides a step-by-step tutorial to learn its features based on incrementally extending an example. In this example programmers build a live multi-user Graffiti application where users can collaborate on a canvas.

Both applications make use of HTML and push-services (web sockets or comet implementations), HTML DSLs, RPC calls, injections and fragments. We show an excerpt of the example in Figure 3.6 and 3.7. The full source of the example (180 lines of code) is available on the project page.

We compare with the code from the Eliom tutorial application, up to “Sending the initial image.” The Eliom source code finishes at 115 lines of code. Our source finishes at 140 lines of code formatted using a production realistic formatting (80 characters-wide). However, Scala is a braces language, and without trailing braces the code is only 124 lines long. The rest of the difference is related to not having better import features, requiring more boilerplate to setup the application (implementing `MTPage`), etc.

This comparison is shallow but we do feel that it gives us enough confidence to say that we have enough features to create web applications without having the codebase inherently suffer from great overhead.

---

7[https://github.com/tzbob/scalagna](https://github.com/tzbob/scalagna)
8[https://ocsigen.org/tuto/manual/application](https://ocsigen.org/tuto/manual/application)
9The last extension does a server-side image rendering, the server libraries used would be too different to make a good comparison.
(* Eliom *)
[%%shared
  open Eliom_content.Html.D (* (1) *)
]

let%shared width = 700
let%shared height = 400

let%client draw ctx ((r,g,b), size, (x1,y1), (x2,y2)) =
  let clr = CSS.Color.string_of_t (CSS.Color.rgb r g b) in
cxt##.strokeStyle := (Js.string clr);
cxt##.lineWidth := float size;
cxt##.beginPath;
cxt##.moveTo (float x1) (float y1);
cxt##.lineTo (float x2) (float y2);
cxt##.stroke

let%server canvas_elt =
canvas ~a:[a_width width; a_height height] []

Figure 3.6: Eliom Draw

// Scalagna
val width = 700
val height = 400

@client def draw(ctx: org.scalajs.dom.CanvasRenderingContext2D) // (2)
  (clr: (Int, Int, Int), size: Double,
   p1: (Double, Double), p2: (Double, Double)) = {
  ctx.strokeStyle = s"rgb(${clr._1},${clr._2},${clr._3})"
  ctx.lineWidth = size
  ctx.beginPath()
  ctx.moveTo(p1._1, p1._2)
  ctx.lineTo(p2._1, p2._2)
  ctx.stroke()
}

@server def canvas_elt =
canvas(widthA := width, heightA := height)

Figure 3.7: Scalagna Draw
4 Functional Reactive Programming and the DOM

During development of multi-tier FRP we identified important trade-offs and design decisions when it comes to an FRP DOM API. This chapter is based on the following publication:


The research was done by me with guidance from Dominique Devriese. The contents of the paper were written by me with helpful remarks and comments from Dominique Devriese and Frank Piessens.

Abstract

Web applications are inherently event-driven and traditionally implemented using imperative callbacks in Javascript. An alternative approach for such programs is functional reactive programming (FRP). FRP offers abstractions to make event-driven programming convenient, safe and composable, but like pure functions it is isolated from the ‘outside’ world. In this paper we describe our experience in developing a library that binds FRP to the document object model (DOM). We describe that in its current state there are fundamental issues that do not yet have a perfect solution. We expand upon the functionality of existing FRP DOM libraries with an FRP model for DOM properties. We show that despite of some design problems a pragmatic library can be created that can be used to create web applications.
4.1 Introduction

Web applications are inherently event-driven and traditionally implemented using imperative callbacks and mutable state. An alternative approach to writing such programs is functional reactive programming (FRP). It offers abstractions to make event-driven programming convenient, safe and composable and it has been successfully applied to the web before [Czaplicki and Chong, 2013, Meyerovich et al., 2009].

FRP has two main primitives: events (a stream of values at discrete times) and behaviors (time-varying values). An FRP application is constructed by composing behaviors and events with a set of FRP operations. However, a program that is only made out of FRP components is like a pure function. For it to be useful it has to be able to interact with the ‘outside’ world. One of those interactions is with the document object model (DOM). In this paper we discuss non-trivial design problems when defining an FRP library for the DOM that do not yet have ideal solutions such as:

**Type of Main?** Entry points of programs are traditionally imperative main functions. In this paper we focus on a declarative alternative, a natural approach for FRP programs. We see two design choices. In the first approach, main has type Html, where Html represents a static HTML tree that may contain dynamic parts, i.e., there are APIs with types like p: Behavior[String] ⇒ Element to build elements with varying context. In the second approach, main has type Behavior[Html] where the type Html now represents an entirely static DOM tree. For example, an application that displays a text behavior:

```scala
val text: Behavior[String] = ...
val main: Html = p(text)
// or
val main: Behavior[Html] = text.map(s ⇒ p(s))
```

In the first case the application is a static p-element with dynamic content of type Behavior[String]. In the second case the application is defined as a behavior of a completely static p-element with content of type String. In section 4.3, we discuss both choices and their consequences in depth.

**Recursion between FRP and the DOM.** Irrespective of the above choice, the DOM tree in an FRP application can change in response to changes in FRP behaviors and events. However, such new elements in the DOM tree may also produce new primitive event streams that represent, for example, button clicks on the event, which the FRP program needs to be able to react to. In other words, there is a cycle of
dependencies between the FRP program’s behaviors and events, the DOM tree that
they define (main) and the primitive behaviors for elements in this DOM tree (e.g.,
click events on buttons).

For example (using the Behavior[Html] approach of main), we create a div that
contains two text inputs. The first text input always mirrors the contents of the
second:

```scala
val content: Behavior[String] = ???
val main = content.map { str ⇒
  div(input(id := "one", tpe := text),
       input(id := "two", tpe := text, str.reverse))
}
```

How do we define content? This is a recursive problem and the solution to it affects
both the purity and the usability of the library.

Pull-based Interfaces to the DOM. When linking the API to the DOM it is
important to think about the underlying FRP library’s evaluation strategy.

The DOM produces events and invokes handlers to represent actions such as user
interactions. But, other than an event producer, the DOM is also a queryable source
of data with for example elements that contain multiple mutable fields such as css
classnames or a user’s input.

Current FRP DOM libraries focus entirely on events and push-driven evaluation.
Reading values from input elements in existing libraries is mimicked by listening to
the appropriate events and rebuilding an in-FRP representation of the actual value.
Not only does this create overhead, it makes the correctness of a behavior rely on the
ability to detect all changes. These libraries for example do not propagate changes
that are made through Javascript property assignments such as:

```javascript
document.getElementById("field").value = "123"
```

This hinders common web-development practices such as enhancing existing form
elements with datepickers or autocompletion engines by adding small Javascript
libraries. The developer is forced to forward the added script’s events to the FRP
system to maintain correctness. In this case, there is actually a known solution
that does not appear to have many downsides, namely push-pull FRP [Elliott, 2009].
However, we know of no system that applies this solution in an FRP web framework,
so it seems worth discussing here.
In this paper we report on our experiences gained while defining an FRP library for the DOM. The library is being developed as part of a project in which FRP is used as multi-tier web-application paradigm, a successor to the ideas that were described in [Reynders et al., 2014]. We identify non-trivial design problems that often do not have ideal solutions. We start with a quick introduction to FRP in section 4.2. Next, we go over the main design problems we encountered while developing our library. The type of main in section 4.3, recursion between FRP and the DOM in section 4.4 and interfacing with the DOM and non-FRP code in section 4.5.

While explaining the design problems and their possible solutions we use an FRP DOM implementation of a to-do list manager as an example. Its functionality is simple and a screenshot is shown in fig. 4.1, a user can add entries and an entry consists of some content and a deadline. All API definitions and code examples are written in Scala but do not use exotic features that would make it hard to port to a different language.

In section 4.6, we finish the to-do list manager example and demonstrate our final API. We conclude in section 4.7 and discuss related work in Chapter 7.

4.2 Functional Reactive Programming

We start off with a small introduction to the underlying FRP library of this chapter. Let us start by going over the FRP primitives, event and behavior:
trait Event[A] {
  def map[B](f: A ⇒ B): Event[B]
  def filter(p: A ⇒ Boolean): Event[A]
  def merge(e: Event[A])
    (f: (A, A) ⇒ A): Event[A]
  def fold[B](init: B)(acm: (B, A) ⇒ B): DBehavior[B]
}

object CBehavior {
  def constant[A](a: A): CBehavior[A]
}

trait CBehavior[A] {
  def map2[B, C](b: CBehavior[B])(f: (A, B) ⇒ C): CBehavior[C]
  def snapshot[B, C](e: Event[B])(f: (A, B) ⇒ C): Event[C]
}

object DBehavior {
  def constant[A](a: A): DBehavior[A]
}

trait DBehavior[A] {
  def map2[B, C](b: DBehavior[B])(f: (A, B) ⇒ C): DBehavior[C]
  def changes: Event[A]
  def snapshot[B, C](e: Event[B])(f: (A, B) ⇒ C): Event[C]
}

Figure 4.2: Event & Behavior API

**Events** can be seen as a sequence of discrete values. Common examples of events are mouse clicks or button presses since they are occurrences that can be timestamped. There are three core operations on events: map, filter and merge as shown in fig. 4.2. We do not discuss map or filter since they behave just like their well-known collection counterparts. merge takes two events (of the same type) and returns an event that fires whenever one of the original events fire. If both events fire at the same time, the given function combines both values into a single new one.

**Behaviors** are values that vary continuously over time. An example of a behavior is the position of the cursor. A mouse is always somewhere but its position may change continuously as you move your hand. The two core operations on behaviors are: map2 and constant as shown in fig. 4.2. constant creates a behavior that never changes its value. map2 has the ability to combine two behaviors with a function. Other convenience functions such as map can be defined in terms of constant and map2.
Discrete Behaviors. While behaviors can theoretically change continuously, they often change at specific times. In these cases we know more about them, we know when they change. This extra information can be exposed to the programmer through a specific interface of discrete behaviors. Other than exposing the time at which discrete behaviors change (def changes: Event[A]), their API is identical to that of continuous behaviors.

Throughout this paper we use an FRP library that has both the discrete (DBehavior) and continuous behaviors (CBehavior) primitives available separately.

Events ⇔ Behaviors Converting from events to behaviors and vice versa is done through two other operations: Event.fold and Behavior.snapshot, also shown in fig. 4.2. Folding an event is similar to folding a list, a starting value and an accumulation function is given to compute a new value whenever a new element arises. Its result is a behavior representing the ongoing accumulation. This behavior changes discretely whenever the event fires. Snapshotting a behavior with an event allows you to inspect the value of a behavior at the rate of that event. The behavior is sampled for every change in the event by applying a combination function to the event value and the behavior’s value at the time. A simpler version of snapshot called sampledBy ignores the event value and simply samples the behavior at the appropriate times.

Design Decisions & Properties. The FRP library that we use throughout the paper and its semantics make a couple of design decisions that differ from other FRP libraries. Most importantly, both discrete and continuous behaviors co-exist as different primitives exposing the push-pull evaluation explicitly as in [Jeltsch, 2011] unlike the push-pull library by [Elliott, 2009]. The library is also entirely first-order, it offers no methods to flatten nested FRP primitives, and this paper and its proposed DOM library do not require them. This makes the FRP library less expressive, but prevents problems with time leaks and makes other primitives such as network transfers easier to implement.¹

4.3 Type of Main?

The traditional entry point of an application is an imperative main function or some given code block. In this section we only focus on declarative alternatives as an entry point.

We describe two designs choices that can be made for a declarative main value and discuss their differences. The examples in this paper use an element constructor

¹See, e.g., [van der Ploeg and Claessen, 2015] for more details on time leaks.
library called 'scalatags'. Instead of writing XML tags to represent elements we use Scala functions with the appropriate name. The DSL supports attributes and children values, for example:

```scala
tag(attributes: AttrPair*, children: (Html || DBehavior[List[Html]])*): Html
```

An element is created using its corresponding tag function, a variable amount of attributes can be set and children elements can be added. Children can be of two types, the first is a simple Html value. It indicates a static amount of children, for example, a tag that is created with three Html children always has three children during the runtime of the program. The second type of children, DBehavior[List[Html]],

4.3.1 Html.

The first design represents main as a static HTML document. This document defines the entire application and dynamic behavior is embedded within. We give it the type Html. We take the todo-list application’s interface as an example:

```scala
case class Entry(content: String, date: String)
def template(entry: Entry): Html = ...
val state: DBehavior[List[Entry]] = ...
val inputForm: Html = ...
val main: Html =
  div(h1("Todo!"),
    inputForm,
    h2("Entries"),
    div(state.map(_.map(template))))
```

We model the to-do entry as a case class with two fields and assume a function template that takes an entry and returns an element that properly lays out its contents. state is a discrete behavior of entries, that is, a discretely changing to-do list. inputForm is the form from fig. 4.1 that contains the text inputs and buttons required to enter entries for the application. Neither implementations matter here, we leave them undefined. With these components we define the main value as a div with a top title, an input form, a subtitle and a list of all the current entries. The element constructor functions have the following type:

```scala
def 'tag'(attributes: AttrPair*, children: (Html || DBehavior[List[Html]])*): Html
```

An element is created using its corresponding tag function, a variable amount of attributes can be set and children elements can be added. Children can be of two types, the first is a simple Html value. It indicates a static amount of children, for example, a tag that is created with three Html children always has three children during the runtime of the program. The second type of children, DBehavior[List[Html]],

---

2http://www.lihaoyi.com/scalatags/
3This is a simplified notation of this function, it is not valid Scala but it can be implemented in several ways, for example, a type-class based approach.
represents the dynamic pieces in the static Html document. It allows for children to be defined as a discrete behavior of elements, a collection that can change its size throughout the runtime of the program. In our example this is shown through the use of div(entries.map(_.map(template))) which creates a discrete behavior with a list of laidout entries.

**Required: Higher-order FRP.** We rely on modeling element children with behaviors to support a dynamic amount of elements in this type of DSL. However, this implies a higher-orderness of the solution, for example, we can define an element that contains a discrete behavior of elements which in turn contain more discrete behaviors of elements. Higher-order FRP is a tricky subject that cannot be implemented without either: (1) recomputing all previous values of any generated behavior or event, (2) making the first-order API more inconvenient (complicating fold etc.) [van der Ploeg and Claessen, 2015] or (3) making the higher-order API more inconvenient by restricting which behaviors may be nested into others [Krishnaswami et al., 2012]. Each proposed solution has its own drawbacks in either usability or API flexibility.

**Implementation.** With a static document representation all dynamic elements are explicitly added. Dynamic behavior is only possible with embedded DBehavior elements. An implementation makes use of this information by attaching and detaching hooks to the DBehavior elements when needed. These hooks call into DOM APIs to create, delete and modify elements as needed.

### 4.3.2 Behavior of Html.

The alternative design has a discrete behavior as its main value. Instead of defining an application as a user interface element, we define the application as a discretely changing value of elements, i.e., DBehavior[Html]. In this case Html is completely static, all changes to the user interface are expressed through the use of DBehavior.

Let us look at the to-do list application again using the same definition for Entry, template, inputForm and state:

```scala
val main: DBehavior[Html] = state.map { entries ⇒
  val eList = div(entries.map(template))
  div(h1("Todo!"), inputForm, h2("Entries"), eList)
}
```

Instead of ‘embedding’ the state behavior in our user interface code, we now map over it. We take the state behavior and based on its entries we create the same interface as before. Showing the current list of entries in the program is now done by
mapping template directly over entries. Note that the constructor functions in this design no longer take DBehavior[Html] elements, in a similar notation as before:

```scala
def 'tag(*@*)(attrs: AttrPair*, children: Html*): Html
```

In this design, it is natural to express a variable amount of children since the main value itself declares that it is a discretely changing value.

**First-order FRP.** An advantage of structuring the type of main this way is that there is no need for a higher-order FRP primitive in the underlying FRP library. All programs are expressed through the regular behavior primitives such as `map2`.

**Implementation.** Representing main as a behavior of Html requires a more advanced implementation. The only knowledge that main conveys is what the interface should look like at certain points in time. It does not define what changed between two interfaces. The implementation has to recover the changes and turn them into actual DOM operations. This phenomenon has gotten popular in practice under the term *Virtual DOM* or *DOM diffing*, for example in Facebook’s React⁴.

### 4.4 Recursion between FRP and the DOM

There is no ‘best’ way to do foreign function interfaces (FFIs) in FRP, but most FRP libraries have an imperative creation of an event source, for example:

```scala
val source = Event.source[Int]
val acc    = source.fold(0)(_ + _)
source.fire(2) // acc is now 2
source.fire(5) // acc is now 7
```

Regardless of the chosen FRP FFI input, there is an additional issue when describing GUIs. As we have seen, the entire application is defined in terms of the user interface with main for example being DBehavior[Html]. The problem is that the user interface also (albeit indirectly) defines the user’s input to the program. An interface contains forms that can be filled in, buttons that can be pressed, etc. This input is then used to define main itself which turns our GUI application into a recursive problem.

A clean solution to this problem could be to come up with a fixpoint primitive. We expect it to look similar to [Devriese et al., 2013] but are not sure of its exact specifications. In this paper we focus on more pragmatic solutions that break the recursion by pre-declaring possible inputs. We discuss two specific solutions, one creates elements upfront to derive events from, the other creates event sources upfront:

⁴https://facebook.github.io/react/
Deriving Events from Elements. As usual we look to the to-do list application for an example, this time its input form and the corresponding submit event:

```scala
val messageInput: Html = ...
val dateInput: Html = ...
val inputForm = form(messageInput,
                  dateInput,
                  button(tpe := "submit", "Add"))
val submit: Event[dom.Event] =
    inputForm.listen(_.onsubmit)
```

The input form is a simple HTML form with two inputs (messageInput and dateInput) and a submit button. Its creation is straightforward and uses the familiar element constructor DSL. Retrieving the submit event from the form is done by calling a `listen` method with a function that says which property you would like to add a callback too. In this case we attach to the `onsubmit` property and create an FRP event `submit` that propagates DOM events whenever the form is submitted5.

This type of input retrieval has some implications on purity. Up to now the element constructor functions were referentially transparent:

```scala
val x = button()
val el1 = div(x, button())
val el2 = div(x, x)
```

`el1` and `el2` would be freely interchangeable, regardless of what code was added. If it becomes possible to derive events from elements, this can no longer be the case. Even though the definition of `x` is exactly the same as `button()`, it would still be observable as a different instance through events:

```scala
val x = button()
val el1 = div(x, button())
val el2 = div(x, x)
val clicks = x.listen(_.onclick)
```

When `el1` is embedded in the user interface the event `clicks` would produce values when the first button was clicked. However, if `el2` is embedded in the interface, `clicks` would produce values when either of the buttons were clicked.

While element constructors become impure, the `listen` method stays referentially transparent since all calls to the same instance return the same event. We trade one point of referential transparency for the other.

Creating Elements with Event Sources. A second implementation of the to-do list’s input form and submit event makes use of event sources:

```scala
5Note that in this solution `inputForm` still cannot be defined in terms of `submit`, a fully recursive solution in this style would require some sort of element references or element builders to tie the knot.
```
val messageInput: Html = ...
val dateInput: Html = ...
val inputForm = form(onsubmit.listen(submit),
  messageInput,
  dateInput,
  button(tpe := "submit", "Add"))

In this implementation, the submit event source is created beforehand. While creating form, or any other element, attributes can be added. An additional attribute method listen binds event sources to certain properties, for example, onsubmit. This API makes the other trade-off, it keeps the element constructors referentially transparent by making the creation of event sources impure.

Modeling a Variable Amount of Inputs  An additional problem is how to model a variable amount of Html inputs. Let’s look at an extension of the example in fig. 4.1 so that entries can also be removed by clicking on them. In the case of the event sources design we use a more advanced version of listen:

val delete: EventSrc[Int] = Event.source
def template(id: Int, entry: Entry): Html =
  div(onsubmit.listen(delete, _ ⇒ id), ...)
val templatedEntries: DBehavior[List[Html]] =
  entries.map(_.zipWithIndex.map(template))

With the extended version of listen it is possible to modify the event value before invoking the event source. We use this to send an identification of the entry that should be deleted, something that is only possible if the event source is known before all entries. If we derive events from elements instead, we get the following code:

def template(entry: Entry): Html = div(...)
val templatedEntries: DBehavior[List[Html]] =
  entries.map(_ .map (template))
val inputEntries: DBehavior[List[Event[dom.Event]]] =
  entries.map(_ .map { entryHtml ⇒
    entryHtml.listen(_.onclick) })

In this case, template remains a simple implementation but entries no longer send to one specific event. Instead the input is modeled as a behavior of a list of events in inputEntries, which again, implies a need for higher-order FRP.

4.5 Pull-based Interfaces to the DOM

Previous sections focused on defining an FRP API that represents the DOM and its actions. In this section we look at how underlying FRP features can affect how close
we can get to modeling all of the DOM’s features.

Applications written against the DOM are inherently event-driven, but elements in the DOM also have properties that can change over time, for example the value of a text input.

So far we saw two ways to retrieve DOM events, either by having a DSL that extracts them from an HTML element, or by plugging sources into elements upon creation. They are all modeled with FRP events, a push-based propagation primitive.

Existing FRP DOM libraries follow the designs we discussed in section 4.4 even when modeling DOM values, see for example: [Meyerovich et al., 2009]. Let us take a look at how a date would be read for the to-do list application. We use the approach from section 4.4, Deriving Events from Elements to demonstrate the push-based read:

```scala
val dateInput: Html = input(tpe := "text")
val date: DBehavior[String] =
  dateInput.read(_.value)
```

The date is read in a push-based manner, the FRP DOM library subscribes to events on `messageInput` and propagates its changes. For each DOM value that is read, event handlers have to be registered and the actual state of the DOM is mimicked in the FRP system. Not only does this create overhead, it makes the correctness of a behavior rely on the ability to propagate all changes. Flapjax for example does not propagate changes that are made through Javascript property assignment (`document.getElementById("field").value = "123"`). FRP DOM libraries that follow this approach are difficult to use with existing Javascript libraries.

A common practice of web developers is to make use of pre-built widgets, for example, a date picker interface. For example, a date field is no longer just a text input, through added interface components a user can now enter a date by clicking through a calendar. The downside is that the underlying text input is no longer changed through user interaction but through the script’s additional interface. This causes values to change that do not have corresponding events. Push-based libraries are not notified automatically and are only correct if the developer adds additional event handler code.

### 4.5.1 DOM Properties as Continuous Behaviors

We propose to expose DOM properties in FRP through continuous behaviors, a *pull*-based primitive. Continuous behaviors do not suffer from the same issues as the push-based approach for property reads. When a value is required, for example, through the `sampledBy` or `snapshot` operations the property is read on-demand. In
the case of the datepicker example, regardless of how the value changes, the correct value will be read on-demand from the original text input.

Much like in section 4.4, there seems to be a trade-off between two methods of linking continuous behaviors to the DOM DSL:

**Deriving Behaviors from Elements.** As an example we implement the to-do application’s input fields:

```scala
val messageInput: Html = input(tpe := "text")
val dateInput: Html = input(tpe := "text")
val message: CBehavior[String] =
  messageInput.read(_.value)
val date: CBehavior[String] =
  dateInput.read(_.value)
```

In this implementation we create two simple inputs using the element constructors. Behaviors are extracted from the user interface elements through the `read` method. It accepts a function that defines which operation should be executed on the element when its value is queried. Just as before, the element constructor functions are no longer referentially transparent when such an API is added.

**Creating Elements with Behavior Sources.** The second example makes use of behavior sources:

```scala
val message: CBehaviorSrc[String] = CBehavior.src(""")
val date: CBehaviorSrc[String] = CBehavior.src(""")
val messageInput: Html =
  input(tpe := "text", value := ")
  .read(message, _.value)
val dateInput: Html =
  input(tpe := "text", value := ")
  .read(date, _.value)
```

In this implementation the `message` and `date` behavior sources are created beforehand. A behavior source can be created through `CBehavior.src` given a default value. The source always returns the default value unless it has been bound to a different value source, for example, an input’s value. Elements are read through a `read` method, it accepts a behavior source which specifies which behavior reads the element and a function to specify how the data should be read. The method returns a new element that represents an element-with-behavior-source.

The trade-off remains similar, creating continuous behavior sources is not referentially transparent. However, if we compare behavior sources with event sources it ‘feels’ less natural. Event sources do not care how they are triggered or to how many elements they are bound. Multiple elements could fire on the same event source
without unintended behavior to creep in. Behavior sources on the other hand can only read from one place. So we are left with a choice, how do we react when a behavior is bound to multiple places? Multiple solutions are possible, we can throw an error at runtime or allow rebinding of the behavior source so that the last bounded element is read, neither are ideal.

### 4.6 Our FRP DOM Library by Example

In the last three sections we discussed design options for an FRP DOM library while using the to-do application as an example. In this section we demonstrate the full implementation using a library that went through the described design process. The library is based on push-pull FRP with $\texttt{DBehavior[Html]}$ as main that makes use of event/behavior sources to create elements that pragmatically solve the recursion problem.

In fig. 4.3, we bring all the pieces together and show the entire implementation. So far we have learned how to model the user interface, the input form and the corresponding behaviors and events. The only thing that we have not yet seen is line 7 to 13 in fig. 4.3, that is, the implementation of state. On line 7 we define `entry`, a continuous behavior that represents the state of the to-be-submitted entry. `submission` (line 9) defines the submitted entry, it is the value of `entry` at the times of `submit`. The application’s state is computed in `state` (line 11) as a list of entries. We create a simple discrete list behavior by folding on `submission`, starting with the empty list and incrementally concatenating new submissions. This completes the entire to-do list implementation.

#### 4.6.1 API

For brevity we do not discuss the complete Scala API implementation. It uses an approach with implicits classes that embeds our API into an existing HTML element solution. Its total API, from a user perspective, is shown in fig. 4.4. All element constructors and attributes are made available by-name as regular Scala values making everything automatically supported in IDEs.

An application that starts through the API has to implement `FrpDomApp` and supply an implementation for `main`. `container` is the HTML element on which the application will be mounted, by default this is the document’s body.
case class Entry(content: String, date: String)
def template(entry: Entry): Html = ...
val message: CBehaviorSrc[String] = CBehavior.src(""")
val date: CBehaviorSrc[String] = CBehavior.src(""")
val entry: CBehavior[Entry] = message.map2(date) { (m, d) ⇒ Entry(m, d) }
val submission: Event[Entry] = entry.sampledBy(submit)
val state: DBehavior[List[Entry]] = submission.fold(List.empty[Entry]) {
  (acc, entry) ⇒ entry :: acc }
val messageInput, dateInput, inputForm: Html = ...
val main: DBehavior[Html] = ...

Figure 4.3: FRP DOM To-do List Manager

def 'tag'(*attrs: AttrPair*, children: Html*): Html
val 'attrName': Attribute
trait Attribute {
  def listen(src: EventSrc[dom.Event]): AttributePair
  def listen[R](src: EventSrc[R],
               f: dom.Event ⇒ R): AttributePair
}
trait Html {
  type El <: dom.Element
  def read[R](src: CBehaviorSrc[R], f: El ⇒ R): Html
}
trait FrpDomApp {
  val container: dom.Element = dom.document.body
  val main: DBehavior[Html]
}

Figure 4.4: FRP DOM API
4.6.2 Discussion

The source code (with minimal differences such as a few name changes) is available online. Feel free to try it, it also contains an extended implementation of the to-do manager application where entries can be deleted through buttons. In our experience it provides a convenient FRP library for the DOM. Aside from the technical benefits of using push-pull FRP, we also found it more natural to use than a push-based model since it is a closer translation to traditional DOM APIs.

4.7 Conclusion

In this paper we discussed several design problems and proposed (pragmatic) solutions. The library is usable but its APIs are sometimes more imperative and susceptible to programmer errors than we would like.

From an FRP implementer perspective this paper provides a summary of some important design decisions that need to be made and the impact they could have. For FRP researchers, we show where there is room for improvement, such as a fix primitive as a safer and easier-to-use solution to the recursion problem.

\(^6\)https://github.com/Tzbob/scalatags-hokko
5 Efficient Functional Reactive Programming through Incremental Behaviors

Applying FRP to a multi-tier web setting where primitives are composed across multiple tiers brings in different challenges compared to the original purpose of animations. This particularly shows in dealing with state that has been build up incrementally, for example, a list of chat messages in a chat application. In this chapter we propose Incremental Behaviors, an additional FRP behavior that encapsulates both why as well as when a behavior changes so that programmers can express incremental values native to FRP.

The research described in this chapter contains the following publication:


The research was done by me with guidance from Dominique Devriese. The contents of the paper were written by me with helpful remarks and comments from Dominique Devriese.

Abstract

Many types of software are inherently event-driven ranging from web applications to embedded devices and traditionally, such applications are implemented using imperative callbacks. An alternative approach to writing such programs is functional reactive programming (FRP). FRP offers abstractions to make event-driven programming convenient, safe and composable, but they come at a price. FRP behaviors cannot efficiently deal with larger, incrementally constructed values such
as a collection of messages or a list of connected devices. Since these situations occur naturally, it hinders the use of FRP. We report on a new FRP primitive: ‘incremental behavior’. We show that the semantics fit within existing FRP semantics and that their API can be used as a foundation for more ad-hoc solutions, such as incremental collections and discrete behaviors. Finally, we present benchmarks that demonstrate the advantages of incremental behaviors in terms of reduced computation time and bandwidth.

5.1 Introduction

Event-driven applications are common in several domains. Traditionally, such applications are implemented using imperative callbacks. An alternative approach to writing such programs is functional reactive programming (FRP). It offers abstractions to make event-driven programming convenient, safe and composable. It has been successfully applied to both GUI programming [Czaplicki and Chong, 2013], embedded devices [Wan et al., 2001], etc.

FRP semantics define two primitives: events (a stream of values at discrete times) and behaviors (time-varying values). Let us introduce these with a small example, an FRP equivalent for the common case of using event handlers to increase a mutable sum:

\[
\begin{align*}
\text{val ints: Event[Int] } &= \ldots \quad /\!/ \text{3, ...} \\
\text{val sum: Behavior[Int] } &= \text{ints.fold}\dagger(0) \{ (x, y) \Rightarrow x + y \} \quad /\!/ \text{0, 3, ...}
\end{align*}
\]

We assume the existence of ints, an event that contains integers. We use the fold\dagger method on events to build up state.\dagger It takes an initial value (0) and an accumulation function \((x, y) \Rightarrow x + y\) as arguments and builds a behavior. The event’s values are accumulated starting with the initial value.

An FRP application is constructed by composing behaviors and events with a set of FRP operations. It typically defines a main behavior to describe the entire application, for example Behavior[UI] as the main value for a GUI application.

While FRP is nice in theory, there are shortcomings that crop up when you use it in practice. This paper focuses on one of those issues.

Computational Overhead.

A practical problem with FRP is that behaviors containing large incrementally constructed values often behave suboptimally, for example a chat view:

\dagger\text{fold}\dagger is marked with \dagger for clarity since a variant named fold is introduced in Section 5.2.
From an event stream of messages (msgs) we accumulate the state of the program (chat). All the messages are concatenated into a list behavior. A view of the state is generated through map by pretty printing all elements.

The problem here is that FRP only keeps track of the complete values within behaviors. It does not keep track of how it changes. In the example above, this means that a change to chat (through msgs) is propagated to chatView as 'there is a new list'. The entire list in the view is then re-mapped every time a new message is added. This makes the occurrence of a new message take $O(n)$ processing time instead of a possible $O(1)$.

This is especially problematic since maintaining large collections in behaviors is common in lots of FRP applications: chat applications have a list of messages, social networks have news feeds, sensor networks have lists of nodes, etc. In practice this means that FRP programmers work around the problem by using events to model concepts that would fit a behavior better, such as representing the chat view not as a behavior, but as an event of added strings.

### Bandwidth Overhead.

Computational complexity is not the only area in which standard behaviors do not perform optimally. Bandwidth intensive operations such as saving a behavior’s history to disk (for logging or debugging purposes) or sending its data across the network, are directly impacted by knowing how behaviors change. The multi-tier FRP-based language as proposed in [Reynders et al., 2014] is a typical example where bandwidth matters. To demonstrate this problem in our chat example, we extend it to continuously broadcast to clients:

```scala
def broadcastToClients(b: Behavior[List[String]]) = ...
broadcastToClients(chatView)
```

In this case, there is no way to efficiently implement broadcastToClients since behaviors cannot express when or how values update. Its only options are to poll for changes followed by either recomputing and transmitting the differences, or by sending the entire new list. In practice this often means that functions similar to broadcastToClients are modeled with less appropriate abstractions such as events:

```scala
def broadcastToClients(init: List[String], changes: Event[List[String]])
```
In addition to reducing computational and bandwidth overhead there are other reasons to express how behaviors change. For example, in an FRP Html library the interface may be modeled as a Behavior[Element]. Compared to completely rewriting the DOM, it would be much more efficient to apply only the changes of such a behavior.

5.1.1 Contributions

To summarize, we make the following contributions:

- We define incremental behaviors and their API and show how they fit within existing FRP semantics.

- We show how our approach is more general than previous work such as incremental collections [Maier and Odersky, 2013, Prokopec et al., 2014] and discrete behaviors [Maier et al., 2010, Salvaneschi et al., 2014, Blackheath, 2015] by implementing them into our framework. Additionally, we show how a joint API between discrete and incremental behaviors based on manually computing differences can form a middle ground between them.

- We present an implementation of incremental behaviors and incremental collections as a Scala library. We demonstrate the advantages of incremental behaviors through a performance analysis of our implementation. In the analysis we compare incremental behavior’s computational and bandwidth overhead with their non-incremental counterpart.

We start by introducing FRP and incremental behaviors in Section 5.2, and we show how incremental collections and discrete behaviors can be implemented on top of their API in Section 5.3. In Section 5.4, we evaluate their performance. Section 5.5 discusses incremental behaviors with respect to higher-order FRP semantics. We highlight related work in Chapter 7 and conclude with future work in Section 5.6.

In addition to our own implementation, we also found an independent implementation of similar ideas in the grapefruit-frp Haskell library [Jeltsch, 2012] (their incremental signals seem similar to our incremental behaviors). This implementation has not been presented in the literature and lacks some of the features described here, but we consider it as additional evidence of the value of incremental behaviors.

5.2 Incremental Behaviors

We present incremental behaviors, an additional primitive for functional reactive programming (FRP). All code examples use our working proof-of-concept Scala
trait Event[A] {
  def map[B](f: A ⇒ B): Event[B]
  def filter(p: A ⇒ Boolean): Event[A]
  def merge(e: Event[A])(f: (A, A) ⇒ A): Event[A]
  def fold[B](init: B)(accum: (B, A) ⇒ B): Behavior[B]
}

trait Behavior[A] {
  def map2[B, C](b: Behavior[B])(f: (A, B) ⇒ C): Behavior[C]
  def map[B](f: A ⇒ B): Behavior[B]
  def snapshot[B, C](e: Event[B])(f: (A, B) ⇒ C): Event[C]
}

object Behavior { def constant[A]: Behavior[A] }

Figure 5.1: Event & Behavior API

implementation and we encourage the reader to play around with it. As a small Scala introduction, in this paper it is sufficient to think of a trait as a Java-like interface, object Foo as a collection of static methods for Foo and the case class A(x: Int) as a data type with field x. Case classes can get constructed (like regular classes) through either new A(0) or just A(0).

5.2.1 Functional Reactive Programming: Event & Behavior

We begin with a summary of FRP and its semantics. We focus on first-order FRP semantics that are very similar to the ones defined in [Jeltsch, 2011, Blackheath, 2015]. For readers interested in higher-order FRP semantics we refer to Section 5.5. Let us go over the two main FRP primitives, event and behavior:

Events are sets of discrete values:

\[ \text{Event}_\tau = \{ e \in \mathcal{P}(\text{Time} \times \tau) \mid \forall (t, v), (t', v') \in e. t = t' \Rightarrow v = v' \} \]

In the denotational semantics above \( \alpha \) is the ‘denotation’ or meaning of \( \alpha \), \( \{ e \in \mathcal{P}(\alpha) \mid P \} \) is the set of elements \( e \) from the powerset of \( \alpha \) for which \( P \) holds. Events are sets of \( (\text{Time}, \tau) \) tuples that do not contain values with duplicate Time components.

Typical examples of these discrete values are mouse clicks or button presses. There are three core operations: map, filter and merge as shown in fig. 5.1. We do not discuss map or filter since they behave just like their well-known collection counterparts. merge takes two events and returns an event that fires whenever one of the original
events fire. When both fire at the same time, the given function combines both values into a single new one.

**Behaviors** can be thought of as values that can vary continuously over time. Semantically, behaviors of type \( \tau \) are regular functions from \( \text{Time} \) to \( \tau \):

\[
[\text{Behavior}_\tau] = \{b \in \text{Time} \to [\tau]\}
\]

An example of a behavior is the cursor’s position. A mouse is always somewhere but its position may change continuously as you move your hand. The two core operations on behaviors are: \text{map2} and \text{constant} as shown in fig. 5.1. \text{constant} creates a behavior that never changes its value. \text{map2} has the ability to combine two behaviors with a function. Other convenience functions such as \text{map} can be defined in terms of \text{constant} and \text{map2}.

**Behaviors ⇔ Events.** Converting from events to behaviors and vice versa is done through two other operations: \text{Event}.\text{fold} \( ^* \) and \text{Behavior}.\text{snapshot}, also shown in fig. 5.1. Folding an event is similar to folding a list, a starting value and an accumulation function is given to compute a new value whenever a new element arises. Its result is a behavior representing the accumulation. Snapshotting a behavior with an event inspects the value of a behavior at the rate of that event. The behavior is sampled for every change in the event by applying a combination function to the event value and the behavior’s value at the time.

The FRP semantics that we just showed make a couple of design decisions that can differ from others: it is first-order instead of higher-order (see Section 5.5), it allows only one event value at a time and behaviors are defined in continuous time as opposed to discrete time (see discrete behaviors in Section 5.3.2).

### 5.2.2 Motivating Example: Todo List

An example FRP program using our Scala library is shown in fig. 5.2. We implement a simple todo list. We leave out most of the code and focus only on the bits that are important to this paper. The user’s intent to submit his message is modeled by the \text{submissionE} event. The state of the todo application itself is created by accumulating all the submissions into a list behavior (\text{todoListB}).

\(^*\)Note that to have a definable semantics for \text{fold} an extra restriction on events (which we omitted for brevity) is required. The occurrences in an event must be ‘uniform discrete’, that is, the amount of events before any time \( t \) must be finite.
case class Entry(title: String, content: String) {
    val pretty = s"$title
$content"
}

val submissionE: Event[Entry] = ...

val todoListB: Behavior[List[Message]] = submissionE.fold(List.empty) { (lst, msg) ⇒ msg :: lst }
val todoListView: Behavior[String] = todoListB.map(_.map(_.pretty).mkString)

def replicate(b: Behavior[String]) = ...
replicate(todoListView)

Figure 5.2: FRP Todo List Example

We create todoListView, a string representation of all the entries in the list by first turning the list of entries into a pretty printed list of strings (_\.map(_\.pretty)) and then concatenating all elements with .mkString.

Without going into details of its implementation, we assume the replicate function that takes a behavior and replicates it to a different application, such as in client/server applications.

This example demonstrates the computation and bandwidth issues for large values that we discussed before. Each new submission accumulates into the application’s state (todoListB), and the mapping to todoListView always recomputes the entire pretty printed string since todoListB does not contain information about how it changes. Furthermore, the replicate function is impossible to implement efficiently since the newly created todoListView behavior does not contain information about how it changes either. replicate has to detect changes and either recompute the differences between two behavior values, or send the entire behavior’s state.

Depending on the amount of submissions, both problems can impact the user experience. A programmer has to either accept an underperforming application or remodel his code with less appropriate abstractions such as events as a workaround.

5.2.3 Incremental Behaviors

The purpose of our new FRP primitive, incremental behaviors, is to capture when behaviors change and how they change. Semantically we interpret them as a triple of an event \( e \), an initial value \( (v_0) \) and an accumulation function \( f \): \n
\[
[IBehavior_{\tau,\delta}] = \{(e, v_0, f) \in [Events_{\delta}] \times [\tau] \times ([\tau] \times [\delta] \rightarrow [\tau])\}
\]
An incremental behavior has two type parameters, $\tau$ denotes the behavior’s value while $\delta$ is the type of its increments. The event component in the semantics refers to the increment responsible for the change in a behavior’s value. The type signature of the $\text{fold}^\dagger$ operation on events in fig. 5.1 is the motivation behind the semantics. From now on we replace $\text{fold}^\dagger$ (which creates $\text{Behavior}$s) with $\text{fold}$ to create incremental behaviors:

```scala
```

Generally, an incremental behavior is a behavior that has been, or could have been, defined using $\text{fold}$. In other words, it can be seen as a refied $\text{fold}$. To work with incremental behaviors we provide the following functions: constant, $\text{incMap}$, $\text{incMap2}$, $\text{snapshot}$ and $\text{toBehavior}$, shown below:

```scala
```

constant and $\text{snapshot}$ work exactly as they do on regular behaviors by creating constants and allowing behaviors to be sampled at the rate of events. The chosen semantics for incremental behaviors make their semantic implementation trivial, but the complexity of folding events is moved to $\text{toBehavior}$.

It executes the fold and turns an incremental behavior into a continuous one ($\text{Time} → \tau$):

$$\text{convert} : \text{IBehavior}_{\tau, \delta} → \text{Behavior}_{\tau}$$

$$\text{convert}(e, v_0, f) = \lambda t. f(f(...(f(v_0, d_1), ...), d_{n-1})d_n)$$

if

$$t_1 < ... < t_n \leq t < t_{n+1} < ... \land \{(t_1, d_1), ..., (t_n, d_n), (t_{n+1}, d_{n+1})...\} = e$$
val todoListIB: IBehavior[List[Entry], Entry] = submissionE.fold(List.empty) { (list, entry) ⇒ entry :: list }
val todoListViewIB: IBehavior[String, String] = todoListIB.incMap { _.map(_.pretty).mkString } { _.pretty }
{ (accStr, dStr) ⇒ dStr + "\n" + accStr }
def replicate(ib: IBehavior[String, String]) = ...
replicate(todoListViewIB)

Figure 5.3: FRP Incremental Todo List Example

`convert` defines a behavior that, upon evaluation at a time \( t \), returns the accumulation (using \( f \)) of all event values up to time \( t \), starting from the initial value (\( v_0 \)).\(^3\) Note that while `toBehavior` is an explicit method in this paper, a subclass relation between incremental behaviors and behaviors is completely reasonable.

`incMap` has the same purpose as a behavior’s `map`, that is, provide a way to apply a function over the data. In the case of incremental behaviors we require three things. (1) \( f \) is the function that maps the old value of an incremental behavior to the new. (2) \( f_δ \) maps old deltas to new deltas, and (3) accumulator tells us how to put new values and new deltas back together. Note that we expect the programmer to take care of a proper relation between the old accumulator \( acc_{old}, f_δ, f \) and the new accumulator \( acc_{new} \):

\[
f(acc_{old}(\alpha, \delta_\alpha)) = acc_{new}(f(\alpha), f_δ(\delta_\alpha)) \quad \forall \alpha \in A. \forall \delta_\alpha \in DA
\]

`incMap2` is also more complex than a behavior’s `map2`. But its purpose is also the same, that is, provide a way to combine two behaviors into one. Its main parameters are a second behavior and a combination function, but two additional parameters are required to produce an incremental behavior. The first, `deltaFun`, takes two values of the incoming behaviors as well as a value of type `OneOrBoth[DA, DB]`. This type contains either an increment of the first or the second behavior (of type \( DA \) resp. \( DB \)) or both. `deltaFun`’s task is to compute an increment of type \( DC \) that represents the change (if any) that the given changes cause in the value of the resulting behavior. The final parameter `accumulator` tells us how to apply the new type of increments to previous values, it is the `fold` function for the new incremental behavior.

**Fixing Todo List.**

We fix the overhead issues that were present in the previous example from fig. 5.2 by using incremental behaviors in fig. 5.3. We omit the creation of `submissionE` since it

\(^3\)This construction assumes that the event fires only a finite amount of times before any fixed time \( t \) (a property we call uniform discreteness).
is identical to the implementation in fig. 5.2.

We create an incremental todo list with incremental state by using fold to create todoListIB. In this example we create an incremental version of the pretty printed todo list (todoListViewIB) by using incMap, the incremental version of map. It takes three arguments. The first defines how to create a pretty printed string from a list of entries by mapping it: _.map(_.pretty).mkString. The second defines how the deltas should change by pretty printing the old delta: _.pretty. The final argument tells us how to combine our new values with the new deltas through string concatenation: (accStr, dStr) ⇒ dStr + "n" + accStr.

The result is a version of the pretty printed todo list that is synchronized with the actual state through a time complexity of $\mathcal{O}(1)$ instead of $\mathcal{O}(n)$. Similarly, replicate can now be implemented efficiently since it has access to a behavior’s fine-grained change. It can directly send just a trace of its changes.

5.3 Incremental Behaviors as a Foundation

The advantages of incremental behaviors are apparent with performance improvements and the ability to model with behaviors where they are appropriate, such as using a behavior for the string representation of our todo list. However, they came at a cost, the API of incremental behaviors is more complex than their non-incremental counterparts. While the general incremental behavior API offers the most freedom, its complexity can be off-putting.

Other work on different FRP primitives such as incremental collections [Maier and Odersky, 2013, Prokopec et al., 2014] and discrete behaviors [Maier et al., 2010, Salvaneschi et al., 2014, Blackheath, 2015] provide similar benefits in certain cases while having a much simpler API. After our general proposal we now demonstrate that these other approaches can be seen as specialized versions of incremental behaviors by implementing them on top of our design.

5.3.1 Incremental Collections

Compared to regular behaviors, it is harder to create composable incremental behaviors. Let us use todoListIB from fig. 5.3 as an example. We create a function toView that takes an incremental behavior of entries and returns a view such as todoListViewIB:

\footnote{To keep the code concise we ignore the inefficient string operations here.}
def toView(ib: IBehavior[String , String]) =
  ib.incMap { _ .map(_.pretty).mkString } { _.pretty }
  { (accStr , dStr) ⇒
    dStr + "\n" + accStr }
val todoListIBF: IBehavior[List[Entry], Entry] =
  submissionE.fold(List.empty) { (list , e) ⇒
    if (e.title.contains("FRP")) e :: list else list
  }

Using toView on todoListIB creates an incremental behavior identical to the
previously defined todoListViewIB. We define a second version of todoListIB that
filters out entries that do not contain FRP within their title (todoListIBF). Using
toView on todoListIBF instead does not create a properly pretty-printed filtered
to-do list version. The problem here is that, although todoListIBF also uses Entry as
the type of deltas, toView fails to take into account the different meaning of the delta
type: a delta of type Entry is unconditionally added to the list in todoListIB, but only
under a certain condition in todoListIBF. This example illustrates a general issue
with incremental behaviors: functions operating on them are now not just coupled to
the representation of data but also to the representation of deltas.

However, for standard types with standard APIs (like collections), we can mitigate this
problem by defining a standard type of deltas (with a standard meaning). Both [Maier
and Odersky, 2013] and [Prokopec et al., 2014] propose incremental collections in a
reactive programming environment to get more efficient collection operations without
adding the API complexity that incremental behaviors bring. For this section, we
focus on [Maier and Odersky, 2013]’s abstraction: an incremental sequence (RSeq[A])
and discuss how it can be implemented on top of incremental behaviors. From a high
level you can think of it as an efficient version of Behavior[Seq[A]]. Its usage and
API is similar to collection libraries as shown in fig. 5.4.

A commonly used operation is map, which for reactive sequences returns a new
reactive sequence. The mapped RSeq does not remap the entire list upon change,
instead modifications to the list are processed on their own. Elements that should
be inserted are mapped separately and their results are inserted directly. The same
goes for deletions, which are propagated to the mapped list and directly remove an

trait RSeq[A] {
  def map[B](f: A ⇒ B): RSeq[B]
  def filter(f: A ⇒ Boolean): RSeq[A]
  def foldUndo[B](init: B)(op: (B, A) ⇒ B)(undo: (B, A) ⇒ B): Behavior[B]
  def flatMap[B](f: A ⇒ RSeq[B]): RSeq[B]
}

Figure 5.4: Reactive Sequence Core API
type IVector[A] = IBehavior[Vector[A], SeqDelta[A]]

sealed trait SeqDelta[+A] {
  def apply(v: Vector[A]): Vector[A]
}

case class Insert[A](element: A, index: Int) extends SeqDelta[A]
case class Remove[A](element: A) extends SeqDelta[A]
case class Update[A](element: A, index: Int) extends SeqDelta[A]
case class Combined[A](d1: SeqDelta[A], d2: SeqDelta[A]) extends SeqDelta[A]

def updated[A](iv: IVector[A], updates: Event[(A, Int))): IVector[A]
def insert[A](iv: IVector[A], insertions: Event[(A, Int))): IVector[A]
def remove[A](iv: IVector[A], deletions: Event[Int]): IVector[A]

Figure 5.5: Reactive Sequence Implementation

element. Other common collection operations work similarly.

We implement reactive sequences as a special kind of incremental behavior: RSeq[A] ≃ IBehavior[Vector[A], SeqDelta[A]]. Our Scala prototype implementation of incremental behaviors also contains a collection library. It implements incremental collections by using a standard delta that models common operations such as addition or deletion. Incremental collection APIs are implemented through incremental behavior operations such as incMap2. It plugs into the Scala standard library and uses the appropriate collection abstractions such as traversable and sequence to provide a generic incremental API for the collection library.

In fig. 5.5, we demonstrate the model of such an incremental collection. Do keep in mind that to avoid Scala-specific concepts we focus on a reactive sequence implementation for just the vector and that an implementation for generic traversables or sequences is more complex.

In short, the incremental vector that we build is a vector data structure that efficiently handles incremental changes based on incremental behaviors. As a first step we model the different types of incremental changes (SeqDeltas). Each increment contains a method apply that defines the application of the increment to a vector, for brevity we assume its implementation. The different types of increments that we support are: an insertion (Insert), a removal (Remove), an in-place update (Update) or a combination of other deltas (Combined). Insertions simply contain the element to be inserted at a specific index, removals contain the element that should be removed and updates contain an element that should replace an element on a specific index.

They correspond to the three functions that are available on the incremental vector: updated, insert and remove as shown in fig. 5.5.
def mapDelta[A, B](d: SeqDelta[A], f: A ⇒ B): SeqDelta[B] =
  d match {
    case Insert(element, i) ⇒ Insert(f(element), i)
    case Remove(element) ⇒ Remove(f(element))
    case Update(element, i) ⇒ Update(f(element), i)
    case Combined(d1, d2) ⇒ Combined(mapDelta(d1, f), mapDelta(d2, f))
  }
def map[A, B](rseq: IVector[A])(f: A ⇒ B): IVector[B] =
  rseq.incMap(v ⇒ v.map(f))(d ⇒ mapDelta(d, f)) {
    (acc: B, delta: SeqDelta[A]) ⇒ delta.apply(acc)
  }

Figure 5.6: Map Implementation

Using the model from fig. 5.5, we implement a simple version of an incremental map in fig. 5.6. Its created by using the incremental behavior’s incMap function. The first argument provides a way to transform the initial vector to an initial result vector. In our case this is a simple map: v ⇒ v.map(f). The second argument contains the transformation on the increments. For this we defined a helper function that applies the function f and use it accordingly:d ⇒ mapDelta(d, f). The final argument defines how new deltas are applied to new values, in our case it remains the assumed apply method on SeqDelta.

For a discussion about higher order APIs on reactive sequences such as flatMap we refer to Section 5.5.

### 5.3.2 Discrete Behaviors

Continuous behaviors change unpredictably and continuously and their semantics are simple because their meaning are functions of time (Time → τ). But, in practice, discrete behaviors are often used [Maier et al., 2010, Salvaneschi et al., 2014, Blackheath, 2015]. The semantics provide less freedom but they express discrete changes. Essentially, they capture and expose when behaviors change. They are represented as an initial value and a stream of value changes:

\[
[DBehavior_\tau] = \{(e, v_0) \in [Event_\tau] \times [\tau]\}
\]

Other than exposing the time at which they change (def changes: Event[A]), their API is identical to that of continuous behaviors.

It turns out that discrete behaviors can be implemented as a special case of incremental behaviors (DBehavior[A] ≃ IBehavior[A, A]) with a trivial implementation for the accumulator. Their simple behavior API is possible since the accumulator never changes and both types are the same, for example an implementation of map:
Incremental Behaviors through Computing Differences.

There are situations where bandwidth is costly such as saving a behavior’s history to disk (for logging or debugging) or sending its data across the network. When time complexity is of less concern, the API complexity of incremental methods (such as `incMap2`) makes them less desirable, regardless of their computational benefits.

To accommodate these scenarios we propose a way to obtain incremental behaviors through the simpler discrete behaviors:

```scala
trait DBehavior[A] {
  def toIBehaviorGeneric[DA](implicit d: Delta[A, DA]): IBehavior[A, DA]
}
```

We recover increments between values by computing their differences. We require `diff` and `patch` functions to be defined with the following relation:

$$\text{patch}(v_1, \text{diff}(v_2, v_1)) = v_2 \quad \forall v_1, v_2 \in A.$$  

`diff` is used to compute differences that work as deltas and `patch` completes the incremental behavior by defining how they fold back into a value. In practice these two functions can often be derived with generic programming approaches. For example, there is a Scala library\(^5\) which could be integrated; in this case the user would see an API like `DBehavior.toIBehaviorGeneric`. Using it is easy, for example, converting rates:

```scala
val rates: DBehavior[Set[Rate]] = ...
rates.toIBehaviorGeneric
```

### 5.4 Evaluation

Accompanying our proposed incremental behavior API is a prototype implementation of our ideas. We evaluate this prototype through some microbenchmarks to confirm our expected results. Note that neither our FRP implementation nor our incremental collection implementation were built with performance in mind and as such, overhead is not as low as it could be.\(^6\)

\(^5\)https://github.com/stacycurl/delta

\(^6\)To make our benchmarks as accurate and fair as possible on the JVM, the measurements are results over multiple VM invocations, each prepared with warmup operations.
Computational Performance.

In fig. 5.7 we demonstrate the results regarding the computational performance of incremental behaviors. This microbenchmark is similar to the todo list example of fig. 5.3. We start with an initial vector of a certain amount of integers (plotted on the x-axis) and map them with a function that generates 10 random numbers. Afterwards, we update a single value in the collection and propagate the change to the mapped collection (time to update plotted on the y-axis). As a base case we use a regular push-based (discrete) behavior that contains a Scala vector from the collection library (marked DBehavior[Vector[Int]]). We compare the base case to three different incremental behaviors. First, a hand coded implementation using the fold primitive, exactly as we did in the todo list example from fig. 5.3 (marked IBehavior[Vector[Int], Int]). Second, an implementation based on the incremental collection abstraction from Section 5.3.1 (marked ICollection[Int, Vector[Int]]). Finally, an implementation which starts from the base case vector and performs a naive diffing comparing all elements of the vectors in order to convert a discrete behavior into an incremental one as discussed in section 5.3.2 (marked DBehavior[Vector[Int]] ⇒ ICollection[Int, Vector[Int]]).

The graphs show that, as expected, the naive approach of the regular behavior is the slowest. It remaps every element in the vector whenever a change is made since it does not propagate what changed. The hand coded example has the best performance, it is hard-coded to handle only one case, modifying the head of the collection. It maps the element in isolation and then creates a new modified Scala vector. Similarly, the incremental collections vector also isolates changes, however it has to go through an added abstraction layer that handles other cases than just changing the head of a collection, this causes a bit of overhead. Finally, the case of recomputing the differences. Recomputing the differences trades remapping for a difference algorithm, depending on the algorithm the outcome may be worth it.

Bandwidth Overhead.

In fig. 5.8 we demonstrate the results of a scenario where bandwidth matters. The goal of the application is to write a trace to disk from which all versions of values can be recomputed. This mimics scenarios where bandwidth is important, such as replicating a behavior’s value across the internet. Its implementation is similar to the todo list example in fig. 5.3. We change one element in the collection several times and after every change we expect to be able to trace a trail of its value on disk. The most efficient implementation logs the first value in its entirety and adds differences afterwards.

We test the same four cases as the last benchmark. We can immediately see that
no matter the vector’s size, the incremental implementations remains constant in storage size since their differences remain identical (1 value change). Again, there is an overhead for the higher abstractions used. Extra information regarding the change to the collection is encoded by the incremental collections vector which implies a larger bandwidth footprint. The hand coded result only supports the specific case of the benchmark and logs the absolute minimum at the cost of added programmer complexity. The base case is a naive implementation that logs the entire vector every time.

5.5 Discussion: Higher-Order Incremental Behaviors

In the paper we used first-order FRP instead of higher order FRP. Concretely, this means that we do not provide APIs that are higher-order in the sense that they work with events of events or behaviors of behaviors. A typical example is:

```scala
def join[A](b: Event[Event[A]]): Event[A]
```

It takes an event that fires new events and returns an event that dynamically includes other events.

Higher-order APIs is a tricky subject because the natural semantics for APIs like `join` cannot be implemented without storing or being able to recalculate all previous values of any generated behavior or event that is incorporated into the network. This implies
CONCLUSION & FUTURE WORK

a very high memory usage, which is known in the FRP literature as time leaks, see [van der Ploeg and Claessen, 2015] for more details. Various solutions have been proposed for avoiding time leaks, often based on restricting which Events may fire from an argument of APIs like join. One particular solution which we like to highlight is [Krishnaswami, 2013], an approach that eliminates such leaks by introducing a new kind of type-based capability that grants the right to allocate memory.

We do not go into more details on higher-order FRP or the ways to avoid time leaks, because we believe incremental behaviors are orthogonal to the problem. If we assume the existence of a naive higher-order flatMap API for standard events (as used by [Elliott, 2009]), then we can implement a higher-order primitive for incremental behaviors. Concretely, we assume a higher-order API with the following function that lets us flatten nested events:

```scala
def flatMap[A, B](e: Event[A], f: A ⇒ Event[B]): Event[B]
```

In this setting, we can implement a higher-order operation for incremental behaviors that looks as follows:

```scala
trait IBehavior[A, DA] {
  private val initial: A
  def incFlatMap[DB, B](fa: A ⇒ IBehavior[B, DB])(
    fb: (A, DA) ⇒ Event[DB])
    (accumulator: (B, DB) ⇒ B): IBehavior[B, DB] = {
    val newDeltas: Event[DB] = this.deltas.flatMap(fb)
    val newInitial: B = fa(this.initial).initial
    newDeltas.fold(newInitial)(accumulator)
  }
}
```

Similarly to incMap, the first argument maps the old values to the new, while the last argument redefines the accumulator. But, the second argument allows new increment events to be inserted (DA ⇒ Event[DB]). In other words, it enables dynamic insertion of deltas into the incremental behavior. Use of incFlatMap is only correct if fa’s behavior and fb’s event have the following relation: ∀a. ∀da. fb(a, da) = fa(applyDelta(a, da)).changes

Interestingly, this incFlatMap on incremental behaviors and join on events is all that is required to implement the missing higher order methods from reactive sequences in Section 5.3.1.

5.6 Conclusion & Future Work

Functional Reactive Programming in theory is nice and simple, but in practice there are some shortcomings that prevent general use. This paper tackles one such problem,
the ability to efficiently deal with large incrementally constructed values.

We presented incremental behaviors, a new FRP primitive that can express *how* and *when* values change. We show that it can be used as a foundation for other abstractions and demonstrate that its benefits are noticeable by executing a performance analysis and comparing it to traditional FRP.

As future work, applying ideas from self-adjusting computations to further automate the writing of incremental programs sounds promising. An automatic and efficient replacement to our `diff` function which converts discrete to incremental behaviors efficiently would be the ideal case.
This final content chapter unites all previous ideas into one web framework that makes it possible to write a client/server web application using just FRP primitives. In this chapter we describe Gavial, a true multi-tier FRP experience that has three tiers (client, session and application) which uses incremental behaviors for efficient tier-crossing primitives, Scala/Scala.JS for a mature multi-tier implementation and a pragmatic DOM API.

The contents of this chapter have been submitted to the Art, Science, and Engineering of Programming, 2020.

The research in this chapter was done by me under guidance of Dominique Devriese. The text written for this chapter was written by me with feedback and comments from Dominique Devriese and Frank Piessens.

Abstract

Developing web applications requires dealing with their distributed nature and the natural asynchronicity of user input and network communication. For facilitating this, different researchers have explored the combination of a multi-tier programming language and functional reactive programming. However, existing proposals take this approach only part of the way (some parts of the application remain imperative) or remain naive, with no regard for avoiding glitches across network communication, network traffic overhead, compatibility with common APIs like XMLHttpRequest etc.

In this paper, we present Gavial: the first mature design and implementation of multi-tier FRP that allows constructing an entire web application as a functionally reactive program. By applying a number of new ideas, we demonstrate that multi-tier FRP can in fact deal realistically with important practical aspects of building web applications. At the same time, we retain the declarative nature of FRP, where behaviors and events have an intuitive, compositional semantics and a clear dependency structure.
6.1 Introduction

Developing web applications requires dealing with the specificities of the web. This includes the distributed nature of applications, partly executing on the client (i.e. the user’s browser), partly on the server, and the different parts communicating over APIs like WebSockets or XMLHttpRequests. Additionally, user input and client-server communication on the web are both naturally asynchronous. These characteristics of web applications have led researchers to design programming languages or frameworks tailored to their development.

Two ideas that have been incorporated in such designs to cope with the web’s distributed and asynchronous nature are multi-tier programming languages and functional reactive programming. Multi-tier languages [see among others Cooper et al., 2007, Neubauer and Thiemann, 2005, Serrano and Prunet, 2016] allow both client and server parts of a web application to be written in a single codebase — offering a joint semantics and eliminating tedious work when developing across tiers. On the other hand, functional reactive programming [Elliott and Hudak, 1997] is an alternative programming model that facilitates development of asynchronous applications and reasoning about their behavior. Instead of using side-effecting callbacks, FRP programs are constructed by composing behaviors and events: components representing time-dependent values.

Some research has explored combinations of FRP with multi-tier. Ur/Web [Chlipala, 2015] allows server-to-client reactive values and Eliom [Balat, 2013] allows client reactive values that get their initial value from the server. However, these approaches only apply FRP in some parts of the application, while others remain imperative. Constructing the entire program as a multi-tier FRP program has been proposed, with primitives for sending behaviors and events to and from the server [Reynders et al., 2014]. However, this design remains naive, with no regard for avoiding glitches, code sharing between tiers, network traffic overhead, compatibility with common APIs etc.

The main contribution of this paper is Gavial: the first mature design and implementation of a multi-tier FRP framework that allows constructing an entire web application as a functional reactive program. We achieve this maturity by incorporating (1) existing ideas: automatic bootstrapping [Reynders et al., 2014], asynchronous FRP [Czaplicki and Chong, 2013], recursive behaviors for building web interfaces [Reynders et al., 2017] and incremental behaviors [Reynders and Devriese, 2017] and (2) introducing some novel ideas: a form of glitch minimization across network communication called tiered glitch freedom (§6.3.6), the three-tier structure of our behaviors and events (§6.3.2), novel support for using XMLHttpRequests (instead of WebSockets) in the absence of server-initiated tier-crossing (§6.3.7), and a mature implementation reusing existing Scala infrastructure and library eco-system (§6.4). Gavial shows that multi-tier FRP for the web requires only thin abstractions on top
of proven technologies. At the same time, we retain the declarative nature of FRP, where behaviors and events have an intuitive, compositional semantics and a clear dependency structure.

Outline  First, we gradually introduce Gavial and its features by incrementally developing a game CircleRoyale (§6.2). Next, we provide more details on Gavial APIs (§6.3) and our implementation (§6.4). We discuss related work in Chapter 7 and conclude in §6.5.

6.2 Multi-Tier FRP by Example

Gavial is an embedded domain specific language in Scala (JVM) and Scala.JS (a mature compiler targeting JavaScript). We introduce it using “CircleRoyale”: a small game shown in fig. 6.1. It is inspired by a trend of simple online multiplayer games such as agar.io where players battle each other on a large playing field with minimal controls. In CircleRoyale, players (shown as circles) continuously move around in the direction of the mouse cursor. They can start attacks by hitting space, spawning a larger flashing circle around the player. It remains active for 2 seconds and then cannot be used for 3 seconds (the cooldown period). The game ends when the player is hit by an attack and the end score is the time spent alive.

One of the qualities of Gavial is that we can easily develop applications incrementally: development steps that would otherwise require large or cross-cutting changes (like making a single-player game multi-player, or changing from a server-pushed to client-pulled communication) are smaller and easier when using Gavial. In this chapter, we demonstrate this by gradually implementing CircleRoyale, introducing and highlighting features of Gavial along the way. We start from a simple single-player single-tier UI, and make small, local and understandable changes towards the

Figure 6.1: CircleRoyale
end result. All code examples are valid Scala and actually running code. Example code is shown inline and detailed API information in captioned listings. We encourage the reader to try out Gavial on http://tzbob.be/gavial, only one command is required to setup a basic project and start the tutorial. Both the inline example (in stages) and the CircleRoyale game are available online: https://github.com/tzbob/circleroyale.

6.2.1 Client Prototype

We start with a working solo-player version of CircleRoyale in which a player can move around and start attacks but there are no opponents.

Functional Reactive Programming We represent user controls as direction and attacking:

```scala
val attacking: DBehavior[Boolean] = -- cut --
val direction: DBehavior[Vec2D] = -- cut --
```

To understand this, let us clarify some terminology. Gavial offers a flavor of FRP that contains: events, behaviors, discrete behaviors and incremental behaviors which we discuss later on. As depicted in fig. 6.2, events are streams of timestamped values and behaviors are time-varying values. Discrete behaviors only change at discrete times, which behave as right continuous step functions. For now, we use only Event and DBehavior, the superscript C refers to computations on the client (see below for other tiers). Events and behaviors have the following API:

**Listing 6.1: Event and DBehavior**

```scala
trait Event[A] { def map[B](f: A ⇒ B): Event[B]
    def fold[B](init: B)(f: (A, B) ⇒ B): DBehavior[B]
    def hold(init: A): DBehavior[A] }

trait DBehavior[A] {
    def map2[B, C](db: DBehavior[B])(f: (A, B) ⇒ C): DBehavior[C]
    def snapshotWith[B, C](ev: Event[B])(f: (A, B) ⇒ C): Event[C]
    def sampledBy(ev: Event[_]): Event[A]
    def snapshotWith[B, C](other: DBehavior[B])(f: (A, B) ⇒ C): DBehavior[C]
}
object DBehavior { def constant[A](a: A): DBehavior[A] }
```

Events can be mapped over with a function or be folded into discrete behaviors. Such folded discrete behaviors start out with the given initial value and “step” to a new
value when the event fires, combining the event’s value and the behavior’s previous value using \( f \). \( \text{hold} \) is like \( \text{fold} \) but simply stores the last event value seen.

Behaviors can be combined using \( \text{map2} \) and can be read at the rate of an event using \( \text{snapshotWith} \) (using a function to combine the values). \( \text{sampledBy} \) is like \( \text{snapshotWith} \) but ignores the value of event \( \text{ev} \). A \( \text{snapshotWith} \) for discrete behaviors works like \( \text{map2} \) but the resulting behavior changes only when other does.

Returning to CircleRoyale, a type \( \text{Player} \) represents a player’s position, and whether he is alive, attacking or dead. The \( \text{svg} \) method produces the player’s circle and optionally the attacking circle as SVG elements.

```scala
case class Player(position: Vec2D, attacking: Boolean, dead: Boolean) {
  def update(direction: Vec2D, attacking: Boolean): Player = -- cut --
  def setDead(dead: Boolean): Player = -- cut --
  val svg: UI.HTML = -- cut -- }
```

We define the player’s state over time, by tupling the user controls (direction and whether an attack should be started) using \( \text{map2} \)\(^1\) and then \( \text{fold} \)ing these changes starting from a default state. For this example we simplify a bit and allow players to attack whenever and for however long they want.

```scala
val input: DBehavior[[(Vec2D, Boolean)]] =
  direction.map2(attacking){ (_, _) }
val player: DBehavior[Player] = input.changes.fold(Player.default) {
  case (p, (dir, att)) ⇒ p.update(dir, att) }
```

**Throttling the update rate**  The user input in direction is based on mouse movement and updates very frequently, so for efficiency, we change \( \text{player} \) to throttle the rate. This is easy to do in Gavial using \( \text{IntervalCycle} \) (an abstraction around JavaScript’s \( \text{setInterval} \)). We sample input by an event \( \text{time} \) which fires at 10Hz and update \( \text{player} \) to use this throttled input instead.

\(^1\)\( (...) \) is a Scala anonymous function that combines two arguments into a pair.
val time: Event[Time] = new IntervalCycle(1.second / 10).elapsedTime
val throttledInput: Event[(Vec2D, Boolean)] = input.sampledBy(time)
val player: DBehavior[Player] = throttledInput.fold(Player.default) {
  case (p, (dir, att)) ⇒ p.update(dir, att)
}

Recursive Behavior  As pointed out before [Reynders et al., 2017], user interfaces are often inherently recursive. For example, in CircleRoyale the future direction of a player is relative to its current position, which is itself determined by the direction of movement. Our flavor of FRP permits explicit recursive definitions (in Scala) through .delayed:

Listing 6.2: Delayed DBehavior

object DBehavior { def delayed[A](db: DBehavior[A]): Behavior[A] }

A delayed behavior will have the same step function as the original behavior but it is left continuous instead of being right continuous as in fig. 6.2. In other words, the delayed version of a discrete behavior keeps the old value for an instant longer when a change occurs. We use this particularly for defining direction.

val svgFRP = new SvgFRP("playground", width, height)
val direction: DBehavior[Vec2D] = {
  val previousPosition: Behavior[Vec2D] = DBehavior.delayed(position)
  val directionEv = previousPosition.snapshotWith(svgFRP.mousePosition) {
    (prevPos, mouse) ⇒ mouse - prevPos
  }
  directionEv.hold(Vec2D.zero).toDBehavior
}
val player: DBehavior[Player] = -- (see before) --
val position: DBehavior[Vec2D] = player.map(_.position)

The player always moves towards the user’s mouse position. We define previousPosition by delaying the player’s position with DBehavior.delayed, obtaining the player’s position just before the current. The current mouse position is retrieved through the SvgFRP object, which represents an SVG tag in the HTML interface and makes screen-to-SVG coordinate conversions. We define directionEv as the difference between the mouse position and the previous position. The final direction behavior is then defined, taking the initial direction as Vec2D.zero.

Whether a player is attacking is simpler: we simply look if the spacebar is down using the keyboard interface.
val kb = new Keyboard()
val attacking: DBehavior[Boolean] = kb.isPlaying(" ")

The Game Interface  A multi-tier FRP application is ultimately created by defining a value `ui` of type `DBehavior[HTML]`. This behavior defines the value of the main HTML tag at every moment, and these values are rendered on screen. In this way, the programmer declaratively defines the application as "everything that is visible to the user". This main value is also a discrete behavior, i.e. it contains a notion of initial value and changes values at discrete times. Simply from this definition, the framework has all the information it needs to efficiently update the client's view.

In our example we want the interface to display the game. We obtain the SVG tag from the `SvgFRP` object by providing a camera and SVG tags. This camera is defined to show a fixed-size view of the game centered around the player. The behavior of SVG tags currently only contains the player's SVG representation. The resulting SVG tag is wrapped up in some HTML to produce our `ui`.

```scala
val camera: DBehavior[Camera] = position.map { p ⇒
  Camera(p - Vec2D(cameraWidth / 2, cameraHeight / 2),
  cameraWidth,
  cameraHeight)
}
val svgContent = player.map(p ⇒ List(p.svg))
val ui: DBehavior[HTML] =
  svgFRP.svg(camera, svgContent).map { svg ⇒
    section(article(svg))
}
```

Note that we use an HTML library that exposes HTML (and SVG) tags as regular Scala functions so that — as is common in multi-tier languages — interfaces are written with the full power of a general purpose programming language.

### 6.2.2 Multiplayer CircleRoyale

In a multi-player version of CircleRoyale, we do not want to compute the state of the world locally on the client. Implementing this change requires surprisingly few changes. In Gavial, going from client to server is easy with tier-crossing primitives `.toSession` and `.toClient`.

Listing 6.3: Primitives for crossing from the Client to the Session tier and vice versa

```scala
object DBehaviorC { def toSession[A](db: DBehaviorC[A]): DBehaviorS[A] }
object DBehaviorS { def toClient[A](db: DBehaviorS[A]): DBehaviorC[A] }
```
The session tier is the server-side counterpart of the client tier we’ve been using so far: its behaviors and events live on the server side and there is one instance of it for each client. Crossing from the client to the session tier simply sends a client’s value to that client’s instance of the session tier, or vice versa. Using these primitives, we change the code as follows (unmodified code in gray).

```scala
val sessionInterval = new ServerTick(1.second / 10).sessionElapsedTime
val serverInput: Event[(Vec2D, Boolean)] = EventC.toSession(throttledInput)
val playerInput: Event[(Boolean, Vec2D)] = serverInput.hold((Vec2D.zero, false)).sampledBy(sessionInterval)
val sessionPlayer: DBehaviorS[Player] = playerInput.fold(Player.default) { case (p, (dir, att)) => p.update(dir, att) }
val player: ClientDBehavior[Player] = DBehaviorS.toClient(sessionPlayer)
```

We send player input to the server using `toSession` and re-throttle it to 10Hz as before, using a server abstraction `ServerTick`. This rate drives the server computation and steps the game forward. With `server-side` player input we define a `sessionPlayer`, with the same logic as before: user input updates the player’s position and whether or not he is attacking. The client-side player definition that we used to define the interface is replaced by simply bringing the `sessionPlayer` to the client tier. These changes are all that is needed to construct a server-side version of the client-side-only code we had before.

**Tiers: Client, Session and Application** However, CircleRoyale is still not multiplayer. To add that functionality, we need a way to combine data from different clients. Gavial offers a third tier where this is possible: the application tier. The conversion functions to the application tier expose the session tier’s multiplicity: a value in the session tier corresponds to a map of values, indexed by `Client`. `Client` values act as `connection` tokens: they identify a browser connection to the server. In the session tier, the `Client` identifier for the current connection is available through the `client` primitive.

```
object DBehaviorS {
  def toApp[A](db: DBehaviorS[A]): DBehaviorA[Map[Client, A]]
  def client: DBehaviorS[Client]
}
object DBehaviorA { def toSession[A](db: DBehaviorA[A]): DBehaviorS[A] }
```

A summary of the three tiers in Gavial is shown in fig. 6.3.
Application Tier and Client Tokens With the application tier, we can add user interaction to our game. We assume an implementation of a pure function `deadClients`, which performs a form of collision detection. It checks among all players that are still alive if one was hit by the weapon of another. We omit the implementation because it is not relevant to our discussion. The method returns all clients that have died (including those within the given players that were already dead).

```scala
def deadClients(players: Map[Client, Player]): Set[Client] = -- cut --
val playerInputAndDead: EventS[[(Boolean, (Vec2D, Boolean))]] =
  DBehaviorS.delayed(dead).snapshotWith(playerInput) { _ -> _ }
val checkedPlayer: DBehaviorS[Player] =
  playerInputAndDead.fold(Player.default) { (p, in) =>
    in match {
      case (dead, (dir, attacking)) =>
        p.update(dir, attacking).setDead(dead)
    }
  }
val checkedPlayers: DBehaviorA[Map[Client, Player]] =
  DBehaviorS.toApp(checkedPlayer)
val losers: DBehaviorA[Set[Client]] = checkedPlayers.map(deadClients)
val dead: DBehaviorS[Boolean] =
  DBehaviorS.toSession(losers).map2(DBehaviorS.client) {_ contains _}
```

Our previous definition of a player is no longer enough: we must now also track whether a player has died. We delay the (soon-to-be-defined) dead behavior and snapshot it with the `playerInput` from before and fold the result to compute a player including its `.dead` property. We then convert our `checkedPlayer` session value to an application value using `toApp`. The resulting behavior contains a map of all connected clients and their respective player states. The collision detection function `deadClients` is mapped over `checkedPlayers` to collect those who have lost the game, both old losers and new. With the set of dead clients, we define `dead` by checking whether or not the session tier’s client (available through `DBehaviorS.client`) is among the dead players.

![Figure 6.3: The tiers available in Gavial.](image_url)
**Drawing Active Players**  At this point, the game is just missing an interface that shows all living players:

```scala
def survivors: DBehavior[List[Player]] = 
  checkedPlayers.map(_.values.toList.filter(!_.dead))

val svgContent: DBehavior[List[HTML]] = 
  DBehavior.toClient(DBehavior[toSession(survivors)].map(_.svg))

val gameUI: DBehavior[HTML] = 
  svgFRP.svg(camera, svgContent).map2(DBehavior[toClient(dead)]) {
    (svg, dead) ⇒ 
    section(article(if (!dead) svg else h1("You died!")))
  }

val ui: DBehavior[HTML] = gameUI
```

First, we filter out the survivors in checkedPlayers and send them to the session, and subsequently the client tier. The result is then turned into a discrete behavior of HTML tags and passed to `SvgFRP.svg`. Additionally, we send the existing dead to the client and use it to show the message “You died!” to dead players.

**Adding a Chat**  Gavial also supports “regular” HTML applications that are not so heavily SVG based. To demonstrate this, we extend CircleRoyale with a minimal chat, positioned underneath the game. Users can type in their name and message and all submissions are shown in a simple list. As soon as a user submits a message, his character is labeled so that people can identify messages’ authors.

A chat message is represented as a `Message`, a Scala case class containing a name and message and a method for converting to a string representation.

```scala
case class Message(name: String, message: String) {
  val string = s"$name says $message"
}
```

```scala
def msgSource: EventSource[Message] = Event(source[Message])
def msgs: Event[Message] = Event.toSession(msgSource)
def appRmsgs: Event[Map[Client, Message]] = Event.toApp(msgs)
def chatInput: Event[List[String]] = 
  appRmsgs.map(_.values.toList.map(_.string))

val chat: DBehavior[List[String]] = 
  chatInput.fold(List.empty[String]) { (acc , n) ⇒ n ++ acc }
```

```scala
def chatUI: DBehavior[HTML] = 
  DBehavior(toSession(chat)).map { c ⇒ 
    ul(c.map(msg ⇒ li(msg)))
  }
```

An event source is created, onto which messages can be pushed. Naturally, the chat is accumulated at the server side, by retrieving messages from the client and folding
them into a list (most recent messages at the top). The accumulated chat is sent back to the client and rendered into an HTML list.

**Incremental Behaviors**  Unfortunately, the current chat implementation has a problem: how chat is sent to the client. This discrete behavior encodes when the list changes but not how. In other words, every time a new message is added to the chat, the chat log is seen as a completely new list of strings. When we send this behavior, this means that the full list will be transmitted to the client on every update, and network traffic will grow over time.

To solve this without using an unnatural encoding of the chat log through events, we provide incremental behaviors [Reynders and Devriese, 2017]. They are behaviors that not only encode when a value changes but also why it changes.

**Listing 6.5: Incremental Behavior**

```
trait IBehavior[A, DA] { def toDBehavior: DBehavior[A] }
object IBehavior {
  def toSession[A, DA](cb: IBehavior[A, DA]): IBehavior[S[A, DA]]
}
```

As shown in listing 6.5, an incremental fold (foldI) on an event creates an incremental behavior. Incremental behaviors can be sent across all tiers, just like other FRP primitives. While they have their own (incremental) operations, for this example it suffices to know that they can be turned into discrete behaviors. We can now replace our suboptimal chat log implementation with a more efficient version, with minimal changes (marked in gray, note the appearance of Is):

```
val chat: IBehavior[A][List[String], List[String]] =
  chatInput.foldI((List.empty[String]) { (acc, n) ⇒ n ++ acc }
val chatUI: DBehavior[C][HTML] =
  SessionIBehavior.toClient(AppIBehavior.toSession(chat))
  .toDBehavior.map { c ⇒
    ul(c.map(msg ⇒ li(msg)))
  }
```

State that accumulates over time is common in event-driven applications. By reifying this as incremental behaviors, they can be sent over the network efficiently.

**Hooking into the DOM**  All that is left to complete the chat is the interface. This allows us to introduce Gavial’s interface to HTML elements and their event handlers:
Listing 6.6: DOM

```scala
object UI { def listen[R](a: Attr, src: EventSource[R]): AttrPair[EventSource[R]]
  (f: js.Dynamic => R): AttrPair[EventSource[R]
} object Event { def source[A]: EventSource[A] }
```

Listing 6.6 shows part of the API for DOM events. Event sources are events with an “open end” and with an imperative API through which non-FRP code can inject values. `UI.listen` takes an extra function which turns a dynamic Scala.js value into a value of type `R` and produces an attribute pair that can be used to install the appropriate event handler on an HTML tag, e.g., `button(width := "5", UI.listen(onclick, src)(_ ⇒ 1))`.

We redefine `ui` a final time. The main value of our application will now show both the game interface and the chat interface. Additionally, it contains a form with a submit button that is hooked up to the `msgSource` event source.

```scala
val ui = chatUI.map2(gameUI) { (chat, game) ⇒
  div(
    game,
    form(
      input('type' := "text", placeholder := "Name", name := "name"),
      input('type' := "text", placeholder := "Message", name := "msg"),
      input('type' := "submit"),
      UI.listen(onsubmit, msgSource) { ev ⇒
        val formElements = ev.target.elements
        val name = formElements.name.value.asInstanceOf[String]
        val message = formElements.msg.value.asInstanceOf[String]
        Message(name, message)
      }
    ),
    chat
  )
}
```

**Player Labels** Finally, we add player labels in game after they have posted to the chat. Although this code does not introduce new functionality of Gavial, it will allow us to explain an important aspect of tier-crossing (see section 6.3.6).
val optName: DBehavior[S][Option[String]] = 
  msgs.map(msg ⇒ Some(msg.name))
  .hold(None).map2(dead) { (n, d) ⇒ if (!d) n else None }

val labelInfo: DBehavior[S][Option[String], Vec2D] =
  optName.map2(sessionPlayer) { (name, p) ⇒ (name, p.position) }

val allLabelInfo: DBehavior[S][List[Option[String], Vec2D]] =
  DBehavior[A].toSession(DBehavior[S].toApp(labelInfo).map(_.values.toList))

val clientLabels: DBehavior[C][List[Option[HTML]]] =
  DBehavior[S].toClient(allLabelInfo).map { ls ⇒
    import UI.html.{svgAttrs ⇒ a}
    ls.map { case (name, Vec2D(x, y)) ⇒
      name.map { str ⇒ text(a.x := x, a.y := y, str) }
    }
  }

val svgContent: DBehavior[C][List[HTML]] =
  clientSurvivors.map2(clientLabels) { (ap, ls) ⇒
    ap.map(_.svg) ++ ls.flatten }

This code collects the name and position on the session tier (for living players who have already posted a message), sends it to the application tier and back (to collect all labels), and next to the client, where the non-empty labels are added to the SVG element.

6.2.3 XHR or Websocket Backend

For implementing the client-server crossing primitives, Gavial can work in one of two ways: using XMLHttpRequests or using WebSockets. The former is more widely supported and requires less resources on the server, but the latter allows bidirectional communication.

CircleRoyale does in fact use bidirectional communication. Consider, for example, the definition of svgContent in the final multi-player example. Remember that survivors is a discrete application-tier behavior of all living players in the game. It updates at a fixed rate of 10Hz and its new value is then pushed to all clients. Because this rate is server-initiated, our implementation requires websockets.

Nevertheless, if we want to avoid web sockets, we can modify the game so that servers do not push values to clients, but clients pull from servers. We already have the client event time, a 10Hz timer event. In polledPlayers, we re-use this time, send it to the server, turn it into a discrete behavior and use that behavior to read out values of survivors as polledPlayers. The polled players in turn get sent back to the client and define the new xhr-compatible clientSurvivors.
val svgContent: DBehavior[List[HTML]] = // unmodified
clientSurvivors.map2(clientLabels) { (ap, ls) ⇒
ap.map(_.svg) ++ ls.flatten }

val polledPlayers: DBehavior[Player] = // xhr-compatible
DBehaviorA.toSession(survivors).sampledBy(EventC.toSession(time).hold(0))

val clientSurvivors: DBehavior[List[Player]] =
DBehaviorS.toClient(polledPlayers)

Note again that no other code needs to change. If we make similar changes for other
server-initiated session behaviors that are sent to the client, our application becomes
xhr compatible. In fact, since both client-to-server and server-to-client updates are
both driven by the time event, messages to the server and responses to the client
can be exchanged in a single HTTP request. We discuss both backends and their
requirements of each in detail in Section 6.3.7.

6.3 Making a Realistic Multi-tier FRP for the Web

After this hands-on introduction to Gavial we take a more detailed look at its main
features. This includes both existing and novel ideas and shows that FRP applied to a
multi-tier web setting can benefit the development of web applications.

6.3.1 Practical FRP and Incremental State

Since there exist quite a variety of FRP flavors in academic literature and in practical
implementations, it is useful to take a moment to discuss where our API can be
situated in the FRP family tree and which changes were made and why, to make it
usable in practice.

We support both discrete and non-discrete behaviors. The latter are behaviors that may
change continuously over time or at unknown times and we do not offer, for example,
a method Behavior[A].changes for them. They are evaluated as needed, similar to
[Elliott, 2009]. This choice allows us to support behaviors that are not native to the
FRP system for which changes are impossible or expensive to track, such as databases
or DOM properties.

On the other hand, discrete behaviors additionally expose when a behavior changes
value. An example where this is useful is the discrete client behavior ui of HTML
tags. Because this is a discrete behavior, the programmer can define when the DOM
should be updated. Discrete behaviors can be converted to general behaviors, simply
by throwing away the “when” information.
Finally, as explained before, we also provide incremental behaviors, which reify the fold operation on events [Reynders and Devriese, 2017] and expose not only when (like discrete behaviors) but also how a behavior changes its value. As shown in the CircleRoyale example, incremental behaviors allow us to implement efficient tier-crossing primitives without forcing programmers to use an unnatural representation of behaviors. They are created by folding events and expose both changes and deltas, the change to the behavior as a result from f on the event from which it was folded, and the value that initiated this change deltas.

```
trait Event[A] {
  def foldI[B](init: B)(f: (B, A) ⇒ B): IBehavior[B, A]
}
trait IBehavior[A, DA] { val initial: A
  val f: (A, DA) ⇒ A
  def changes: Event[A]
  def deltas: Event[DA] }
```

They can be converted to discrete behaviors by dropping the why information of deltas. However, discrete behaviors can also be treated as a special case of incremental behaviors. In this case, the changes match the deltas and the folding function f simply ignores the older value while using the new value: (_, a) ⇒ a). We use this property frequently to re-use incremental behavior specific APIs.

**First-order FRP** We also limit our FRP to first-order FRP (as opposed to higher-order FRP). In other words, we do not offer APIs like `flatten: Behavior[Behavior[A]] ⇒ Behavior[A]` that flatten nested FRP abstractions. First-order FRP is conceptually simpler, because dependencies between behaviors and events are statically known. These guarantees make it suitable for multi-tier FRP, as dynamically generated client/server crossing would be hard to understand and implement. It also avoids certain tricky problems of higher-order FRP, like the so-called time leaks that cause memory leaks in naive higher-order APIs (see, for example [van der Ploeg and Claessen, 2015]), and we do not need to modify our API to prevent them. On the downside, first-order FRP is less expressive, but as shown in [Winograd-Cort and Hudak, 2014], the full expressiveness of higher order FRP is not always necessary.

### 6.3.2 Tiers

Our API is tailored to the standard web distribution model, where there is essentially one server and an arbitrary number of active clients (browsers) that connect to the server. We assume that these clients are only active for a subset of the application’s lifetime and we distinguish the programs in multiple tiers. Previous work in multi-tier languages in general [Cooper et al., 2007, Neubauer and Thiemann, 2005, Serrano and
Prunet, 2016, Chlipala, 2015, Balat, 2013] or specifically in multi-tier FRP [Reynders et al., 2014] work with two tiers: client and server. A problem with a two-tiered system is that the framework makes a decision to focus itself to one style of programs. Regular request-response interaction between the client and the server is easier if the chosen server tier is most akin to the session tier. Typical create-read-update-delete applications fall in this category. On the other hand, applications that rely heavily on user-to-user interaction through the server are more difficult to write and have to imperatively manage state across clients somehow. Such interactive applications are easier to write if the server tier is most akin to the application tier since sharing state across clients is part of the programming model as in Chapter 2. However, programs that primarily focus on handling a single client’s requests become very tedious to write.

As such, we support both types of programs in an equally convenient way through the three tiers previously explained and illustrated in fig. 6.3: the application (single instance, server-side), session (client-specific, server-side) and the client tier (client-specific, client-side). The server keeps track of every active client connection and assigns it a unique identifier. This value is exposed in the API as opaque values of type Client and shows up in certain tier-crossing primitives (e.g., .toApp) as well as in the following primitives:

```scala
object Behavior5 { val client: Behavior5[Client] }
object Event4 { val clientChanges: Event4[ClientChange] }
object IBehavior4 { val clients: IBehavior4[Set[Client], ClientChange] }
```

The client primitive exposes a session’s Client as a session behavior. The application event clientChanges informs about clients connecting or disconnecting. We use Scala’s sealed traits to encode the event information (the Client and whether it just connected or disconnected):

```scala
sealed trait ClientChange { val client: Client }
case class Connected(client: Client) extends ClientChange
case class Disconnected(client: Client) extends ClientChange
```

### 6.3.3 Crossing Application & Session Tier

When sending events or behaviors between the session and application tier, the primitives need to deal with the fact that the session tier exists in many copies at the same time (one for each active client), while there is only one instance of the application tier.
This is reflected in the type of the session/application tier-crossing primitives for events:

```scala
object Event^S[A] { def toApp(e: Event^S[A]): Event^A[Map[Client, A]] }
```

Sending a session event to the application tier produces an ApplicationEvent for a different type of values: Map[Client, A] instead of A. Intuitively, the event .toApp(e) will fire whenever at least one of the copies of the session event e fires and a map will be produced containing the identifier of the connection and the event value for each of these copies. Conversely, an application event e can be sent to a session event of type Event^S[A] where it fires for each client whenever e fires.

The situation is similar for sending behaviors.

```scala
object Behavior^S[A] {
  def toApp(b: Behavior^S[A]): Behavior^A[Map[Client, A]]
}
object Behavior^A[A] {
  def toSession(b: Behavior^A[A]): Behavior^S[A]
}
```

Sending a session behavior to the application tier creates a Behavior^A[Map[Client, A]] which maps active clients to their value of the behavior. The converse primitive simply produces a behavior with the same value for every client.

### Incremental & Discrete Behaviors

Sending incremental behaviors between the session and application tiers is a bit more complicated. We do not discuss discrete behaviors, they can be seen as a special case of incremental behaviors and all techniques discussed here are valid for those.

Consider first a session incremental behavior b and think about what the type of IBehavior^S.toApp(b) should be if b has type of values of A and type of deltas DA. As before for regular behaviors, the type of values for Behavior^S.toApp(b) should naturally be Map[Client, A]: a map containing for every active client the value of the corresponding copy of the session incremental behavior. But now we should choose a type of deltas that can represent any way in which the value initiates change. Obviously, one or more of the copies of b may change, so we need this type to contain Option[ClientChange], that
is, each delta is a map of client specific changes (possibly empty) and a possible change in client connections. Both deltas can also appear at the same time, for example, if the value delta is derived straight from the `clientChanges` primitive. The complete type of the method:

```scala
object IBehavior⁵ {  
def toApp[A, DeltaA](sb: IBehavior⁵[A, DeltaA]):  
  IBehavior⁴[A, DeltaA]  
    (Map[Client, A],  
       (Map[Client, DeltaA], Option[ClientChange]))
}
```

The other way around, sending an application incremental behavior to the session tier simply sends values and deltas directly:

```scala
object IBehavior⁴ {  
def toSession[A, DeltaA](sb: IBehavior⁴[A, DeltaA]):  
  IBehavior⁵[A, DeltaA]
}
```

### 6.3.4 Crossing Client & Session Tier

Being able to send events and behaviors from a client tier to a server tier is one of the key features of our model. We do not offer tier-crossing primitives for non-discrete behaviors between the client and session tier. The reason is that we want to use only asynchronous communication between client and server. Imagine a tier-crossing primitive for a client (non-discrete) behavior `b` to the server side as `b.toSession`. It is generally impossible to predict upfront at the client side when the value of `b.toSession` will be required on the server, so the only possible implementation would have the server synchronously request the current value to the client and block execution until the client answers.

The client/session tier-crossing primitives need to transmit values across the network. This requirement shows up in the API as type-classes encoded as Scala’s implicit arguments [Oliveira et al., 2010, Odersky et al., 2017]. All values that cross the network are required to be serializable, visible in our API as extra requirements on the type in form of: `A: Encoder: Decoder`. We use an existing Scala library to supply encoders and decoders for standard items and for case classes (semi-)automatic derivation is available.

Between the client and session tier, sending **events** is straightforward. A client event `e` of type `Event⁵[A]` can be sent to the server as `.toSession(e)` of type `Event⁴[A]` and vice versa using `.toClient(e)`. Intuitively, when the event `e` fires, it asynchronously
sends to the other end of the tier boundary. At that other end, the event fires from
the sent event after the network delay when its received.

```scala
object EventC {
    def toSession[A: Decoder: Encoder](e: EventC[A]): EventS[A]
}
object EventS {
    def toClient[A: Decoder: Encoder](e: EventS[A]): EventC[A]
}
```

For an incremental behavior \(b\), it is known when the value changes, so we can
produce correct values at the other side of the network for the sent behavior
\(\text{.toSession}(b)\) or \(\text{.toClient}(b)\) simply by sending an update whenever \(b\) changes.
As explained before, this does not require transmitting the full value of behavior \(b\),
but just the deltas that represent what has changed. We can re-compute the new value
on the server from the previous value and the delta. In other words, if \(b\) has type
\(\text{IBehaviorC}[A, \text{DA}]\), we only need to transmit the value of type \(\text{DA}\) when \(b\) changes
and the result is an incremental behavior of type \(\text{IBehaviorS}[A, \text{DA}]\). Of course, when
the client first connects, there is no point in transmitting a delta and we transmit the
full initial value.

```scala
type En = Encoder
type De = Decoder
object IBehaviorS {
}
object IBehaviorC {
}
```

### 6.3.5 Bootstrapping Clients

One of the useful properties of combining FRP with multi-tier languages is automatic
bootstrapping of clients [Reynders et al., 2014]. Bootstrapping is the initial
provisioning of client values with the latest state of session behaviors sent to the
client. It is a standard task in web application development, typically solved in an
application-specific way by for example, embedding initial values in the HTML or
by polling for the latest values at client startup. The multi-tier FRP abstractions of
Gavial allow us to solve the bootstrapping in a general, natural and transparent way.

This property is a direct result of the (natural) semantics of \(\text{.toSession}\) on incremental
behaviors, which define the initial value of an incremental session application behavior
b as the value of b at the connection time of the client. If an application behavior sent
to the session tier is further sent to the client, the client will also be provisioned with
this value. This saves developers the work of implementing manual initialization
schemes and automatically helps them define the initial state of new clients.

6.3.6 Tiered Glitch Freedom with Minimal Overhead

Something that we have not discussed before are the guarantees of the tier-crossing
primitives and how they differ from regular FRP semantics. Correctly implemented
FRP libraries follow FRP semantics and protect programmers from partial event
propagation. For example, you would expect t in the following expression to remain
true throughout updates to x:

```
val x: Behavior[Int] = -- cut --
val y: Behavior[Int] = x.map(_ + 1)
val t: Behavior[Int] = x.map2(y)(_ < _) // true
```

Propagating x = 20 from x = 1 should evaluate 20 < 21 for b instead of ever ending
up in 20 < 2 or 1 < 21. If such partial updates can be observed, they are called
glitches. Our proposed multi-tier FRP has a similar property for network-crossing
primitives. To explain this, we visualize propagation of the above toy example as a
graph in fig. 6.4 and add a network between x and y, and t. This corresponds to the
following code:

```
val x: DBehavior[Int] = -- cut --
val y: DBehavior[Int] = x.map(_ + 1)
val t: DBehavior[Int] =
  DBehavior.toSession(x).map2(DBehavior.toSession(y))(_ < _)
```

![Figure 6.4: x < y across tiers](image)

In a naive implementation of multi-tier FRP (such as [Reynders et al., 2014]) glitches
would inevitably occur in t due to network delays, however in our proposal neither
client to server communication nor server to client communication results in glitches\(^2\). In Gavial, all events or behaviors that cross from the client to the session tier and vice versa are propagated atomically. This means that \( t \) will always be true, no partial updates can ever reach the session tier.

Glitches are still possible, but only if FRP values depend on values of different tiers, for example as follows:

\[
\begin{align*}
\text{val } t &: \text{DBehavior}^{\text{c}} \, [\text{Int}] = \\
& \text{DBehavior}^{\text{s}} \, . \, \text{toClient} ( \text{DBehavior}^{\text{c}} \, . \, \text{toSession} (x)) \, . \, \text{map2} (y) (\_ < \_) \\
\end{align*}
\]

In this case, there will be a delay between the updates of \( y \) and the version of \( x \) sent to the server and back. We purposefully do nothing to hide such network delays, as that would require large synchronisation overhead across the network, but leave it up to the programmer to take network delays into account. In other words, we prevent those forms of glitches that can be prevented without working around the distribution model of the web. We will refer to this property as “Tiered Glitch Freedom”. It is not the purpose of this paper to express or prove this property formally, but it is in fact expressed by our denotational semantics (see section 6.3.9).

Perhaps surprisingly, we rely on this property in two cases of the CircleRoyale example. Once in the interaction of defining direction and showing all players on the user interface and another time in the interaction between naming players and showing them. In both of these cases we are sure that: (1) a player’s direction is calculated based on the position that is actually shown in the playing field and (2) a name in the chat is always drawn at the same time on the playing field. There can be no consistency mismatches between names, direction or what is visible in the playing field even though we conceptually do three separate tier-crossings: sessionPlayer (for direction), clientSurvivors (for drawing players) and name (for labeling drawings). Our tier-crossing primitives guarantee that all values cross the network atomically within one propagation cycle. This property allows programmers to more safely refactor and add to existing multi-tier FRP code with the added guarantee that data does not get propagated in unexpected ways. With a naive multi-tier FRP implementation we would have had to go back and manually batch updates into a single .toClient call. This same property is available in the other direction as well, all toSession crossings are done in the same atomic manner.

Tiered glitch freedom gives us strong guarantees: manipulating behaviors or events on one tier and then sending to another is equivalent to first sending and then manipulating them. For example, the previous definition of \( t \) behaves identically to the following definition:

\(^2\)With the exception that it is possible to make use of the FFI or Async to intentionally separate network propagation for efficiency.
val t: DBehavior[Int] = DBehaviorC.toSession(x.map2(y)(_ < _)) // true

Our approach avoids a large implementation cost (as seen in Section 6.4) but still provides useful guarantees. As such, it forms a middle ground between two extremes:

**Naively Connecting FRP Applications**  
Client/server web applications are usually treated as separate programs and previous work on multi-tier FRP also treats client and server programs essentially as distinct but connected FRP applications [Reynders et al., 2014]. Instead of having consistency guarantees regarding glitches, cross-tier connections are treated as a communication channel that lies beyond the scope of the FRP semantics. In practice this means that events are transmitted efficiently, as soon as they come in but the guarantees we have for $t$ do not hold.

**Distributed Reactive Programming**  
The other extreme alternative to our tier-crossing semantics is glitch-freedom in a distributed setting. [Drechsler et al., 2014, Margara and Salvaneschi, 2014, Myter et al., 2019] Essentially, this work completely abstracts away the network entirely and creates the illusion of a single FRP application running across different nodes. In other words, the communication primitives no longer have any visible delay. As far as FRP semantics are concerned everything happens instantly. This comes with a hefty cost. For example, if input validation is done on the server, then these semantics require that input from the client is transmitted to the server, validated and transmitted back, all in the same timeslot, so that the client and server both block until everything is finished. Additionally, some proposals require additional middleware or synchronization protocols that move away from the web architectural model. In summary, this approach has perhaps the simplest semantics and validates a stronger notion of glitch-freedom, but it does not meet the efficiency requirements that we set for programming web applications.

### 6.3.7 XHR or WebSockets?

A novel aspect of Gavial is that it automatically selects the network communication backend to use based on the primitives used to write the program. The xhr mode can be used as long as the application does not require the server to initiate communication with a client. WebSocket mode becomes a requirement as soon as functionality cannot be implemented in a request-response style manner. Intuitively this happens in two cases: (1) whenever a server sends something to a client on its own (through timers or through the foreign function interface) or (2) whenever clients send information to other clients through the server.
To decide whether such cases are present, Gavial analyses the FRP graph and tags every event or behavior as “needing bidirectional communication” or not. Operations such as map simply take the mode of the parent event or behavior. Operations that combine multiple events or behaviors such as map2 take the most restrictive mode of its dependencies, if one requires bidirectional communication then so does the result.

An exception to this rule are the snapshotWith operations. Snapshotting a behavior b that requires bidirectional communication with an event e that does not, produces a result that does not require bidirectional communication either. This makes sense because changes in b will not cause changes in b.snapshotWith(e).

Calculating whether or not the bidirectional communication is necessary is done at startup time and developers can place asserts to force xhr-mode. This means errors do not show up during compile time, but they are reliably detected during development.

### 6.3.8 Interacting with the World

We have seen a glimpse of how to interact with the world in Section 6.2. However, most of the actual interaction was hidden behind some convenient abstractions such as SvgFRP. While these foreign APIs are not part of the core design effort, they make Gavial realistic and practical.

**Connecting to non-FRP APIs** There are three main ways of interacting with non-FRP APIs from within the FRP system, through event sources, by polling behaviors and through behavior sinks:

```scala
Listing 6.7: Imperative FRP API

object Event {
  def source[A]: EventSource[A]
  def sourceWithEngineEffect[A](eff: (A ⇒ Unit) ⇒ Unit): EventSource[A]
}

object Behavior {
  def sink[A](default: A): BehaviorSink[A]
  def fromPoll[A](f: () ⇒ A): Behavior[A]
}

trait Engine {
  def fire(pulses: Seq[(EventSource[A], A)
    forSome { type A }]): FireResult
}
```

Event sources are “open” events. They have the added ability of being triggered imperatively through an “engine”. The engine is an exposed value of the underlying FRP library and contains a fire method that starts a propagation cycle in the FRP network. Another way of making an event source is through the sourceWithEngineEffect method, this requires a function that gets a function as
a parameter of type $A \Rightarrow \text{Unit}$. The given function imperatively fires a value onto the event source that is created through the method and allows programmers to conveniently write code that interacts with the DOM, for example an excerpt of the Keyboard class:

```scala
def keyEvSrc(name: String): EventSource[Key] =
  Event.sourceWithEngineEffect[Key] { (fire: Key ⇒ Unit) ⇒
    @client val _ =
    dom.window.addEventListener[KBEvent](name, ev ⇒
      if (!ev.repeat) fire(ev.key))
  }
```

In this case the Scala.js DOM APIs are used to attach an event handler to the top window object. The handler uses the `fire` function to send keypresses directly to the event source that is being made. `@client` is an annotation that is required to use Scala.js specific APIs in the multi-tier section of a program, more about this in section 6.4.1.

For behaviors there are two options to interact with the outside world: polling behaviors and behavior sinks. A behavior created through `fromPoll` creates thunks around a function. A thunk is created on every propagation cycle and is forced whenever a value from the behavior is required, inside one propagation cycle a `fromPoll` behavior always returns the same value. A behavior sink is very similar except that it makes the polling function re-settable. As long as no poll-function is set it returns the supplied default, it is heavily used to add property support in the DOM API [Reynders et al., 2014].

### Builtin DOM Support

The DOM API incorporates techniques discussed previously by Reynders et al. [Reynders et al., 2014]. We give a brief overview, for a more detailed explanation on the DOM API and its design decisions we refer to that work. In the CircleRoyale example only half of the UI API is used the full API supports listening to DOM events and reading from DOM properties:

```
Listing 6.8: DOM

object UI {
  def listen[R](a: Attr, src: EventSource[R])
    (f: js.Dynamic ⇒ R): AttrPair[EventSource[R]]
  def read[R](tag: HTML)
    (sink: BehaviorSink[R], selector: js.Dynamic ⇒ R): HTML
}
```

Listening to events in the DOM is done by creating additional attributes with `listen`. These special attributes are created with a function and an event source where the
given function \((f)\) takes a dynamic Scala.js value and transforms the DOM event to a concrete result that has to match the type of the given event source. DOM Events are propagated to the FRP program as long as the special attribute is attached. Not just events are supported, by placing behavior sinks on an HTML tag it is also possible to read from DOM properties. The \texttt{selector} function is used to read from the element into the sink and similar to \texttt{listen}, properties are read as long as the special tag is in use.

**Asynchronous FRP** For now, our FRP system executes single-threaded. However, we support Elm’s asynchronous FRP [Czaplicki and Chong, 2013] which allows the programmer to break out of ordered event processing and enable concurrent execution within FRP programs.

Listing 6.9: Async FRP

```scala
object Async { def execute[A](ev: Event[IO[A]]): Event[A] }
```

Through \texttt{Async} it is possible to execute an \texttt{IO[A]} and retrieve an \texttt{A} in a different propagation cycle on the resulting event. We use a library implementation of the \texttt{IO} monad for Scala which has both a JavaScript and a JVM implementation and allows developers to communicate asynchronously with external services such as other web APIs or databases without blocking the FRP program.

### 6.3.9 Denotational Semantics

While we hope that the API is intuitive and easy to comprehend, it is of course important to specify the semantics of the API completely and precisely. We defined a denotational semantics for Gavial as a non-ambiguous reference specification of the core APIs. Time and network delays were modeled and the semantics were actively used during API design. They helped us get the types of the tier-crossing APIs right and gave us useful insight when dealing with corner cases (particularly related to bootstrapping, see section 6.3.5). The denotational semantics do not play a large part in this paper and we do not use them to prove novel properties, but they were helpful as an implementation specification and might be helpful as a reference to the reader, it is available as supplemental material on http://tzbob.be/gavial/semantics.pdf.
6.4 Implementation

6.4.1 Embedded as a Library in Scala

Gavial is completely embedded in Scala in order to use existing libraries and it makes use of Scala.js [Doeraene, 2013], a Scala to JavaScript compiler. It is set up using the same techniques used in the Scalagna project [Reynders et al., 2018], an experimental multi-tier-as-a-library for Scala. Gavial is implemented as two Scala libraries: a JVM and JavaScript library, as well as some common shared code. The server and client-side FRP primitives are respectively backed by real and mock implementations on the server and vice versa on the client and we make sure tier-crossing primitives are supported appropriately on each side.

There are several layers of implementation to Gavial as shown in fig. 6.5. There are JavaScript (blue), JVM (red) and shared (blue & red) sections for their resp. platforms. A shared code section defines code that is included in both platforms.

At the top of Gavial there is an API definition that defines all primitives for a single tier, these include events, behaviors, tier-crossing primitives, etc. This definition has two non-platform specific implementations: FRP and Mock.

An FRP implementation of a tier implements the API with an FRP library. Events and behaviors actually work and cross-tier dependencies are passed through FRP primitives. A Mock implementation on the other hand implements nothing but multi-tier dependency tracking, the FRP primitives and their operations are essentially null-ops.

Both JVM and JavaScript libraries make use of the Mock and FRP tier. Note that the Client tier should do nothing on the JVM while the Application tier should not do anything in JavaScript. Both these tiers are implemented using the Mock tier while the FRP tier is used to implement the others.

From a Gavial user’s perspective, a program is a cross-build between two environments — the JVM and JavaScript backends — which is supported in the Scala build tool (SBT) using plugins. Since Gavial is just a library this means that a developer gets nice integration into known production quality Scala tools in comparison to creating a new multi-tier language from scratch, for example a build system and IDE support as shown in fig. 6.6.

Reusing the Scala/Scala.js Ecosystem  Developers can also make use of the entire Scala/Scala.js ecosystem of libraries as well as the JVM and JavaScript ecosystems through the corresponding Scala FFIs. Libraries that are only supported for one backend can be integrated, either in a backend-specific source file, or by using special
(and somewhat crude) annotations `@client` and `@server`. `@client`-annotated code is not compiled on the server and vice versa.

### 6.4.2 Efficient Tiered Glitch Freedom

As we have seen, client-server communication should behave with minimal glitches. To achieve this, all sent behaviors and events have a unique identifier, which is the same in the client and server code. To cross tiers with tiered glitch freedom we merge all events and behaviors into one large funnel event on the sender side. On the server-side, it is of type `Client ⇒ List[Message]` and contains a function that for all clients produces an (optionally empty) list of messages. Messages consist of the tier-crossed event or behavior’s identifier and a value. At the receiver’s side, a message router receives this list of messages, splits it up into updates that can be fed into the local FRP engine in a single propagation cycle. This whole process is shown in fig. 6.7. This shows that implementing tiered glitch freedom is low-cost both in implementation as well as in performance cost in an FRP program.

![Figure 6.5: Structure of the Gavial implementation.](image)

![Figure 6.6: Auto Completion for Gavial in IntelliJ](image)
**Backends**  The exact implementation depends on the backend the program is running on. The websocket backend simply uses bidirectional communication as expected. Whenever a propagation cycle ends on one end the changes are sent to the other and vice versa. The xhr backend works a bit different since client propagation cycles take care of both directions of communication. At any time the client propagation cycle requires events to be propagated, a request is sent to the server. The server running in xhr mode then executes a propagation cycle which creates new values that should be shipped back to the client. This entire process is synchronous and the client immediately gets its results back as a response to its request.

**Performance**  A small test in our student lab on CircleRoyale (the full version on websockets) was run on one PC while continuously adding players to the game. On an unoptimised single-threaded version of CircleRoyale, we were able to sustain 35 concurrent clients, i.e. $\approx 350$ client-to-server messages and $\approx 2100$ server-to-client messages per second. The most taxing operations were in collision detection (naive implementation) and the underlying messaging library. This limited test indicates that Gavial does not impose a large overhead compared to the underlying tried-and-tested Scala libraries.

### 6.4.3 Crossing Tiers with Incremental Behaviors

Propagating changes of behaviors is entirely similar to those of events. However, extra support is needed to replicate initial values. Similarly to merging all changes into one big event, behaviors are all bundled into a single behavior of type $\text{Client} \Rightarrow \text{List[Message]}$.

The data to properly initiate the incremental behaviors is sent when a client connects to the server. Exactly when and how depends on the backend. Inwebsocket mode the initial data is pushed from the server to the client as soon as a connection is made. In xhr mode a request is sent from the client as soon as the client-side program is loaded.
6.5 Conclusion and Future Work

In this paper, we have focused on the idea of multi-tier FRP (specifically for the web’s client-server architecture) with asynchronous tier-crossing primitives. Several existing and novel ideas in both FRP and multi-tier research fit together to form Gavial.

The core API and primitives (FRP with crossable tiers) were enriched with asynchronous behaviors, APIs to work with imperative programs, HTML support, etc., to support real-world applications. Novel ideas such as a three-tier model, minimal glitches and support for XHRs are available as a library to be used in the matured Scala toolchain. While the main emphasis of Gavial is to provide a mature and usable web programming framework, a formal semantics specifying the exact behavior of its main APIs is available (see supplemental material).

While Gavial has matured, we have plenty of future work in mind. While the FFI already allows us to access external tools like databases, we would prefer to have more in-FRP APIs. We would like to build an in-FRP database API and benefit from multi-threaded execution of the server tiers without the explicit use of Async. We also want to support deployment schemes that support multiple physical servers without requiring advanced persistent reverse proxies. Some ideas include using an in-FRP database API for persistent storage in combination with a scalable in-memory database backend for the application tier.
7 | Related Work

In this chapter, we describe work that is related to the previous chapters.

To minimize repetition, we summarize related work of both Section 7.1 and Section 7.2 in Table 7.1. These two sections are closely related since they both target work on multi-tier programming languages, their work that is closely related to Chapters 2 and 3. In Section 7.1, we go over the first five columns of the table and look at reactive programming features of several multi-tier languages. We provide details regarding tier-crossing primitives and their properties. In Section 7.2 we discuss multi-tier languages and their implementation and pay close attention to languages which were implemented on top of an existing base language (corresponding to the final column of the table).

The two remaining sections are not related to multi-tier programming and are treated separately. We discuss work specifically related to Chapters 4 and 5. Section 7.3 describes applications of functional reactive programming to the web on the client side in particularly related to GUIs. Finally, Section 7.4 describes work on incremental functional reactive programming semantics and incremental computing.

7.1 Multi-tier Reactive Programming Languages

In this section we discuss distributed reactive and/or multi-tier programming languages and relate them to our work. We do not go into detail on multi-tier language proposals that do not have reactive programming features such as the initial proposed multi-tier calculus by Neubauer and Thiemann [2005] or later additions such as [Choi and Chang, 2019] as well as ML5 by Vii et al. [2007]. The overlap between the languages would be very little and it wouldn’t do their work justice.

Regarding multi-tier languages, we look at languages that are based on existing languages such as Eliom [Radanne et al., 2016] (OCaml), ScalaLoci [Weisenburger et al., 2018] (Scala), Hop [Serrano et al., 2006, Serrano and Queinnec, 2010] (Scheme),
Table 7.1: Comparison Table for Multi-tier Reactivity: green → has feature; red → does not have feature; gray → not applicable; other → in table

In the field of distributed reactive programming there are several programming languages or even algorithms that describe systems relevant to our multi-tier implementation of FRP with tiered glitch freedom such as SID-UP [Drechsler et al., 2014], DREAM [Margara and Salvaneschi, 2014] and QPROPd [Myter et al., 2019].

We focus on whether or not there is support for reactive programming on the client, the server, both or on flexible tiers¹ (also written as both). For these flexible projects we specifically look at the availability of reactive tier-crossing primitives and the consistency properties thereof.

Single-language (and program) web development as showcased by Neubauer and Thiemann [2005] was originally implemented in several projects both in industry as Google Web Toolkit [Google, 2006] as well as in academia from scratch in Links [Cooper et al., 2007] and with Scheme as a base language in Hop [Serrano et al., 2006]. It tackles the distributed nature of the web by providing a language in which a web program can be written as one whole.

¹Some multi-tier languages support distributions other than the client/server architecture of the web.
Reactive programming in turn addresses the asynchronous and event-driven nature of web applications. As such, several of these multi-tier languages have reactive features.

### 7.1.1 Local Reactive Programming in Multi-tier Languages

The most common place for reactive features is at the client-side of the multi-tier language, where the asynchronous nature of the web is most prevalent due to frequent user interaction.

Both Hop (based on Scheme) and its successor HopJS (based on JavaScript) have reactive programming libraries named HipHop(JS) [Berry and Serrano, 2014, Vidal et al., 2018] respectively. The HipHop libraries are based on synchronous programming languages such as Esterel by Berry and Gonthier [1992] and make it possible to create reactive programs in a synchronous DSL similar to Esterel. Synchronous programs are written in isolation and plug into the regular Hop execution as input to output event processors. HipHop supports both execution on client and server-side of Hop but does not provide any means to create one conceptual reactive program across tiers and thus does not provide a means for automatic bootstrapping nor any cross-tier reactive consistency guarantees.

Ur/Web provides a source that can be compared to an EventSource that we discussed throughout the dissertation. It has the same imperative functionality as references, you can create them and set or get its value. Only creation or setting the source is supported on the server. Composing sources is done by “subscribing” to a source and creating a signal. Such a signal allows composable reads over several sources and can be embedded into Ur’s HTML pages. This gives developers an imperative RPC-style interface from the server to a client-side FRP program. The entire page is created from the current source values and as such Ur/Web has a similar elegant solution to the bootstrapping problem we describe in Section 2.2 and Section 6.3.5.

### 7.1.2 Multi-tier Reactivity

Embedding reactive programming in multi-tier programming by making it possible to write reactive programs in each tier is a first step. Several languages go further (like we do) and allow building a reactive program that spans all tiers with primitives to cross tiers. We divide related work in three sections of multi-tier reactivity, those that provide local glitch freedom, total glitch freedom and tiered glitch freedom.

**Local Glitch Freedom**  We discussed a first version of local glitch freedom in Chapter 2. We provide .to(Client/Server) on events and discrete behaviors, but
naively connect client FRP applications to server FRP applications without minimizing glitches or providing any consistency guarantees.

Eliom provides a client/server reactive abstraction (since v5.0). They provide a client *signal* that can be initialized on the server and used on the client. They also provide a server *signal* and have similar tier-crossing primitives that naively propagate events from one tier to the other. As such, they provide a similar solution to the bootstrapping problem as well as a multi-tier reactive programming environment similar to Chapter 2.

An extension of AmbientTalk/R to combine the advantages of loosely-coupled publish/subscriber systems with the elegance of reactive programming constructs is explained in *Loosely-Coupled Distributed Reactive Programming in Mobile Ad Hoc Networks* by Carreton et al. [2010]. They provide *ambient behaviors* which is a construct that allows the propagation of events to reactive values hosted on other FRP networks by means of publish/subscribe. An *ambient behavior* is a behavior that is subscribed to previously exported behaviors. Our approach can be compared to theirs by looking at `to(Client|Server)` as a combination of export/subscribe. Since we assume a ‘single server with multiple clients’ architecture we greatly simplify our API, as a result we do not provide the flexibility that AmbientTalk/R provides.

Flask is not a multi-tier language applied to the client/server web, it is a distributed FRP language for sensor networks. They provide support for broadcast topologies and have no consistency guarantees regarding propagation.

**Total Glitch Freedom** An “ideal” reactive multi-tier program maintains glitch-freedom in all possible configurations, however with network delays involved this comes with a performance compromise.

Other than specifically targeting web development, academia also focused on a more general distributed reactive programming (DRP) with the aim of providing alternatives to the Observer pattern in a distributed environment. An overview of requirements and challenges of DRP is provided in *Towards Distributed Reactive Programming* by Salvaneschi et al. [2013]. The projects we compare with in this space are not multi-tier languages specifically targeted toward the client/server nature of the web. They are targeted towards a larger distribution pattern of a reactive program where, e.g., multiple distributed reactive expressions make up a single program. In our distributed multi-tier project it is about how to unify a client and a server reactive program. Nonetheless, the programming models they build and propose are very related to our multi-tier reactive programming.

[^2]: [https://opam.ocaml.org/packages/eliom/eliom.5.0.0/]
Margara and Salvaneschi [2014] defines a DRP approach that focuses strongly on consistency guarantees. They deliver three levels of consistency guarantees: causal, glitch free and atomic. Causal consistency refers to propagation that maintains causality within one process, e.g., $e_1$ happens before $e_2$ in the origin reactive nodes and will only be able to be observed in that order by other reactive nodes. Glitch free consistency means that a partially propagated FRP network is never observable, even in the distributed setting. Finally, atomic is a consistency guarantee that delivers total FIFO ordering and glitch freedom and thus is the most expensive of them all. Their implementation for glitch free consistency (including atomic, which adds distributed locking to it) requires cross tier propagation messages to include extra details (the history of the propagation) which causes the network traffic to increase. While their consistency guarantees are flexible, they do not provide a consistency guarantee that is similar to tiered glitch freedom.

Several other distributed reactive algorithms were proposed with similar goals. Drechsler et al. [2014] proposes SID-UP, a distributed glitch-free propagation algorithm that minimizes messages compared to DREAM and requires a centralized “lock” that make the distributed program unable to process more than one propagation at a time. Myter et al. [2019] proposes QPROP, an algorithm that provides distributed glitch-free propagation that does not require a central coordinator for locking.

ScalaLoci is not a multi-tier programming language applied to the web, however, it is very related to our work. They also target the Scala language and also do this without modifying the compiler. Instead of having two (or three) set tiers they provide a type system in which a programmer can express the distribution of the program. The placement types are used to define on which location certain expressions live and they support reactive programming with tier-crossing primitives. The consistency guarantees of these tier-crossing primitives are flexible and pluggable, so far they support SID-UP and a propagation similar to [Meijer, 2010] which provides no distributed guarantees. It seems a version of tiered glitch freedom would complement the project well.

**Tiered Glitch Freedom** Ur/Web provides client-side reactive programming but also has an interesting consistency property for its server-to-client tier-crossing primitives. There are no formal semantics on the Ur/Web RPC calls but if we understand correctly, Ur/Web provides a consistency property similar to tiered glitch freedom in the direction of server-to-client. Ur/Web’s programming model is tied tightly to the request-response style of the web, all server-to-client communication within a response of a client-to-server RPC call is done atomically.

In comparison, our work has the same consistency guarantee, but instead of only providing it in one direction, our tiered glitch freedom (see Section 6.3.6) has the same guarantee in both directions.
**Incremental Propagation** None of the multi-tier reactive languages and algorithms we described have support for incremental propagation. They have no primitives similar to our `toClient/Session` for incremental behaviors. As such, incrementally built behaviors such as an incremental collection propagate their full state instead of their change.

### 7.2 Multi-tier Languages and their Implementation: Comparing to Scalagna

We can divide multi-tier languages into two large categories. The first are those that are built from scratch such as Ur/Web [Chlipala, 2015] or Links [Cooper et al., 2007]. Such languages can be tailored to the domain of web programming. For example, Ur/Web closely ties its garbage collection to HTTP requests. However, these languages start out with a fresh ecosystem and often lack the rich library that existing languages have to offer.

The second category of multi-tier languages, like Scalagna, are based on existing languages. Hop [Serrano and Queinnec, 2010], is based on the Scheme language. It is made up of two compilers: one compiles code executed by the server, the other compiles code executed by the client. While Scheme has existing libraries, it’s not the most popular language with web developers. Hop’s successor, Hop.js [Serrano and Prunet, 2016], aims to solve this by using JavaScript as its main language. While still being run through its own compiler to allow server parallelism, the project also focuses on compatibility with existing Node.js libraries. In Hop.js, services are created using their own keyword in the language and compared to our limited use of fragments it allows true full stage programming to define client programs within other client or server programs.

Eliom [Radanne et al., 2016] is based on OCaml and its semantics were a big inspiration to all the decisions made in Scalagna. Its features are very similar, services are created through functions and client and server sections are annotated per declaration. It is more mature than Scalagna and has a better theoretic foundation. Its fragments are truly server values, e.g., a fragment function can be applied to a fragment parameter.

Haste [Ekblad and Claessen, 2015] is based on Haskell, the developers created a GHC to JavaScript compiler and provide a library approach to multi-tier programming. It has no need for client or server annotations. It uses a client and server monad, since there are no untracked side-effects all pure code can be compiled for both versions of the program. The server monad is executed server-side and the client monad defines the client program. It has the issue of not being able to effectively deal with libraries that only exist in the ecosystem on one tier.
JS-Scala [Richard-Foy et al., 2013] is a Scala DSL to write embedded JavaScript programs which was used to create the initial prototype of multi-tier FRP in Chapter 2. It allows multi-tier programming by treating the Scala program as a web-server and a program generator. However, its support for JavaScript and Scala library interop is much less than Scala.js.

7.3 Reactive Programming for the Client Side Web

Applying FRP to web applications is not new, in this section we use the term FRP loosely and describe mostly reactive DOM APIs.

Flapjax [First, provided an FRP implementation for client-side web development in the form of a library and as a standalone language [Meyerovich et al., 2009]. Flapjax is a push-based DOM library which allows elements to contain behaviors. They do not have one explicit main value but allow the embedding of their HTML components, an approach that is similar to having multiple DBehavior main values.

A more recent web language that was based around FRP is Elm [Czaplicki and Chong, 2013]; it is mainly focused on GUI development, but recent versions of the language no longer rely on FRP. By looking at how their interface library changed we can see that they tried both imperative solutions for the recursion problem that we discussed, first going with impure creation of input elements and then switching to an event source. Elm always had a push-based DOM API without abstractions to read properties from the DOM.

UI.Next [Fowler et al., 2015] is an F# DOM library that focuses on providing a higher-order API for the DOM. They have a DBehavior main value and split the API in two layers, a data/flow layer and a presentation layer to tame the higher-order APIs.

In practice there are several other libraries that offer a reactive API for the DOM, for example React[^3], a Javascript library that uses a virtual dom approach to implement a component-based interface framework that provides declarative rendering. Its API is far from FRP. A more FRP-centered approach is Reflex-DOM[^4], a Haskell library similar to UI.Next. It is based on the Reflex FRP library and allows (modified) higher-order primitives and follows an imperative monadic style of binding FRP to the DOM.

[^3]: https://facebook.github.io/react/
[^4]: https://github.com/reflex-frp/reflex-dom
7.4 Incremental Behaviors

We split this related work section into two categories: (1) functional reactive programming and its semantics, and (2) incremental computing, and more specifically self-adjusting computations (SAC).

**Functional reactive programming (FRP)** In contrast, our work focuses on extending the first-order semantics with support for explicit incremental computations. The incremental behaviors that we propose are a generalization of patterns that appear in other work like incremental lists [Maier and Odersky, 2013, Prokopec et al., 2014] or discrete behaviors [Maier et al., 2010, Salvaneschi et al., 2014, Blackheath, 2015]. An implementation of incremental behaviors that is, as far as we can tell, close to the semantics proposed in this paper can be found in the grapefruit library [Jeltsch, 2012], but its semantics are not written down in documentation or an accompanying paper.

**Incremental Computation (IC)** IC is a way of implementing programs that do not redo entire calculations after a change in input. We divide this work based on how much manual work a programmer has to do.

Some approaches based on memoization [Pugh and Teitelbaum, 1989] and self-adjusting computations (SAC) [Acar et al., 2006] minimize manual interference by using dependency graphs and propagation algorithms to efficiently react to input changes. In practice, this has several similarities with FRP, which is also frequently implemented by propagating changes through dependency graphs. However, their focus and granularity differ. SAC focuses on efficiently reacting to small changes in input while FRP focuses on providing simple denotational semantics for event-driven programs. Our FRP work with incremental behaviors is a middle ground which allows programmers to manually express a finer granularity compared to traditional FRP. It allows incremental computations that use the FRP implementation’s propagation and dependency tracking to be defined by the user. [Cai et al., 2014] describes a different approach to automated incremental computations. They define ILC, a static and extendable program-to-program transformation that lifts incremental computations on first-order programs to incremental computations on higher-order programs.

A different approach to automated solutions are frameworks that help programmers with writing incremental computations. Instead of trying to automate the encoding of incremental algorithms it aims to make such computations easier to write such as the reactive sequences [Maier and Odersky, 2013]. We discuss two other examples in detail:
i3QL [Mitschke et al.] proposes relational algebra as a suitable API for incremental computing. i3QL operators are a high-level abstraction for incremental computing implemented on top of a low-level Observable-like change-propagation framework that supports events for adding, removing and updating values. Incremental behaviors are an addition to traditional FRP to further capture when and how values change, it’s more akin to the lower-level implementation of i3QL’s propagation framework (which supports ’when’ and ’how’ natively). It would be very interesting to see to which extent incremental behaviors and FRP can be used to implement i3QL’s relational algebra operators.

Firsov and Jeltsch [2016] describe a general framework in which incremental changes and their propagation are made to compose through typeclasses. They define a typeclass for changes:

```haskell
class Change a where
  type Value p :: *
  ($$) :: p -> Value p -> Value p
```

A change of some type has a value to which it can be applied using the $$ function. Incremental operations can be defined as transformation from one type to another. A transformation in its simplest form is a pair of two functions:

```haskell
data Trans p q = Trans (Value p -> Value q) (p -> q)
```

The similarity between these transformations and our incMap on behaviors is interesting. incMap takes three functions as arguments, one turns a state A into another, a change DA into another and finally a function that defines how the new change and the new state can be combined. This corresponds exactly to the Trans and Change functionality. Trans defines how to convert both values and changes from one type to another while Change makes sure that there is a way to apply changes to a value.

Note that they propose several more complicated versions of Trans, e.g.:

```haskell
data Trans p q =
  forall s. Trans (Value p -> (Value q, s)) (p -> s -> (q, s))
data Trans p q =
  Trans (forall s. Value p -> ST s (Value q, p -> ST s q))
```

The first provides access to a local pure state. Something that can be emulated with incMap. The second version provides safe access to local mutable state, something for which we do not have an alternative. Given the similarities between the two approaches a logical step for future work seems to incorporate the typeclass-based framework for incremental computing into the incremental behavior API which could give us alternative definitions such as:

```haskell
trait ITBehavior[A: Change] {
```
def incMap[B: Change](trans: Trans[A, B]): ITBehavior[B]
}

As a final note, several SAC papers [Acar et al., 2013, Hammer et al., 2015] refer to FRP and point out opportunities to combine the two, for example, [Acar et al., 2013] says “Although FRP research remained largely orthogonal to incremental computation, it may benefit from incremental computation, because computations performed at consecutive time steps can be similar.” Incremental behaviors may be a stepping stone towards a better integration of incremental computations in FRP.
8 | Conclusion

In this final chapter, we look back to consider our contributions. The initial goal is to investigate the possibility of using FRP primitives to compose an entire client/server web application. In doing so we achieved results in both the fields of multi-tier languages, FRP as well as multi-tier FRP. We discuss the limitations and look towards further potential directions.

8.1 Multi-tier Scala

In Chapter 3, we design a library that make it possible to use Scala and Scala.JS as a multi-tier language. As shown in section 3.2, programmers write client and server annotations to denote where an expression is supposed to be computed. Anything that is not annotated is duplicated and run on both client and server. Client sections get compiled with Scala.JS and server sections go through the regular Scala compiler.

Something that was surprisingly powerful is the re-use of existing ecosystems through the annotations. Both Scala and Scala.JS have a set of libraries that work only on their systems (and of course, those that work on both). These annotations make sure that packages that are only available on one section do not cause errors in the compilation run of the other section, e.g., a Scala.JS DOM library does not cause errors in the server compilation as long as it is only used in client annotated expressions. As explained in section 3.4, this is all possible as a library with macros. No changes were made to the Scala or Scala.JS compiler.

While not the primary result of our research, it is an interesting side-result of our work. However, it has some limitations. The annotations cannot be nested, in contrast to other multi-tier languages and the re-use of server or client specific libraries is clunky since the macro system of Scala does not allow import statements to be annotated.
Future Work  The annotation macros are implemented in ≈100 lines of code and simply do two things: error if annotations are nested or delete the expression contents depending on the tier. To be able to do this expression deletion we make use of an experimental macro feature. At the time of writing, it does not seem that the future versions of Scala (3.0) will contain similar features, making our multi-tier Scala annotations out-dated. An alternative version would have to be written as a compiler plugin. This makes it a bit harder to distribute (but still possible using just a couple of lines in the build file).

However, we could be more ambitious. Scala is in a unique situation to develop an actual multi-tier language since all the hard work has already been done. Both compilers are mature and there is a thriving ecosystem of server-side web development libraries as well as client-side libraries. A true Scala multi-tier implementation would instantly be on par with industry standards such as Google Web Toolkit in terms of library availability.

8.2 Efficient FRP

In Chapter 5, we discuss incremental behaviors, our addition to the FRP primitives. We provide denotational semantics that supplement the FRP denotational semantics naturally, incremental behaviors are triplets of event, initial value and folding function. They model incremental event handling, that is, computations that happen as a result of a new event. In section 5.3, we show them to be a more general form of representing incremental reactive data types by implementing incremental collections and discrete behaviors. Other than possible computational efficiency, incremental behaviors also expose why a behavior changes its value. This extra information makes it possible to efficiently perform operations where bandwidth is important, e.g., saving a log to disk incrementally or sending a list across the network.

Limitations of our incremental behaviors are a need for first-order FRP. In section 5.5, we discuss possible higher-order implementations of incremental behaviors.

Future Work  While there is an overlap between incremental computing and our incremental behaviors, incremental computing focuses on making it easier to write incremental algorithms. Incremental behaviors do not help with this, it only makes these algorithms native to FRP. They will still be hard to write and will still require much more manual effort. It would be interesting to see how techniques from incremental computing could be applied to automatically turn computations on normal or discrete behaviors into incremental behaviors, making programs easy to write and efficient to execute (e.g., with a trade-off in memory).
8.3 FRP and (Multi-tier) Web Development

The main contribution of this research is the application of FRP to web development. Not just to make the interface easier to write, but to compose the entire application, client through server.

We succeeded with a two-step process, first we created a prototype in Chapter 2 and made sure the ideas were viable. We identified all shortcomings that made it unrealistic to use for practical web applications and found novel solutions such as incremental behaviors. In Chapter 6, we created a web framework named Gavial that combined existing research with novel ideas to become a three-tiered framework where developers compose FRP primitives between three tiers to create a web application.

**Future Work** While we succeeded in showcasing that it is possible to write a client/server web application in FRP, we still rely on a foreign interface to interact with necessary components outside of the “FRP-world”. The main thing that comes to mind is a database. Much like we did with the DOM API we would like to design a database FRP API. We imagine that select queries can be abstracted over with behaviors and insert, update and delete queries with events.

8.4 Potential Directions

Each chapter itself had our thoughts on future work, but in this section we look at the bigger picture. Given the results as they are now, what more is possible?

FRP and multi-tier programming both share a similar advantage, that is, more information is made concrete and is visible to the programmer explicitly. There are no hidden and untyped or unchecked connections between a server and client in multi-tier languages and there are no side-effecting state values scattered throughout the program in FRP. While this is an advantage of its own, it can be exploited to a greater extent in tooling. A multi-tier FRP debugger that uses all of the extra explicit information available can show information to programmers that normal debuggers could not.

Gavial runs in a threading system comparable to Node.js. It has one event loop in which the main FRP code is run and all code expecting to block this loop should be run asynchronously in a different thread with results being fed back. However, our three-tier system of client, session and application provides an opportunity to do more. Session tiers could be implemented in threads isolated from the (global)
application tier. Since this information is explicit to programmers it makes the choice of using (harder-to-scale) application-wide state concrete and more visible. With adjustments to the tier-crossing primitives between session and application tier the implementation can be made to run session in their own threads with synchronization between an application thread. This would make multi-tier FRP be even closer to how web applications are written today.
A | Denotational Semantics

This document contains the denotational semantics of ‘Gavial: Programming the web with multi-tier FRP’.

First, we define our semantic domain, secondly we list all core operations regarding clients, events and behaviors. Our project has three kinds of behaviors (plus events) on three tiers, in order to cut down on the size and make this document as legible as possible, our core operations section does not contain the written semantics for all tiers. Instead we limit ourselves to the client tier, the basic operations have no tier-specific abnormalities and the definitions for the session and application tier are nearly identical. For examples of written semantics on core operations on a different tier we have used application tier core operations in the paper.

Thirdly we introduce the conversion primitives. All operations that do not have an impact on time (i.e., no delays) but convert from one tier to another or from one FRP abstraction to another are given.

Next, we define the boundary operations which give precise semantics to server-to-client and client-to-server primitives.

Finally we finish up by proving some useful properties.

A.1 Domain

We define the domain of our denotational semantics. We begin with definitions of all building blocks such as client identifiers and status and time.

\[ \text{Client} \subseteq \mathbb{N} \quad \text{finite}(\text{Client}) \quad [\text{Client}] = \text{Client} \]
\[ \text{ClientStatus} = \{ \text{Connected, Disconnected} \} \]
\[ \text{ClientChange} = \text{ClientStatus} \times \text{Client} \]

The start and end of both the entire application as well as a specific client exists.

\[ \text{Time} = \mathbb{R}_{\geq 0} \]
\[ \exists \text{Time}_0, \text{Time}_\infty \in \text{Time} \]
\[ \forall c \in \text{Client}. \exists \text{Time}_{0,c}, \text{Time}_{\infty,c} \in \text{Time} \]

We define server slots, server slots combined with regular time form the time as seen on the server. That is, the time as observed by the server is indexed by connections to a client. In practice this corresponds to multiple client connections to a server that are processed in sequence.

\[ \text{ServerSlot} = \text{Client} \uplus \top \]
\[ \text{Time}^* = \text{Time} \times \text{ServerSlot} \]
\[ \forall s_1, s_2 \in \text{ServerSlot}. s_1 < s_2 \iff (s_1 = \top \land s_2 \neq \top) \lor (s_1 \neq s_2 \land s_1 < s_2) \]
\[ \forall t_1, t_2 \in \text{Time}. \forall s_1, s_2 \in \text{ServerSlot}. \]
\[ (t_1, s_1) < (t_2, s_2) \iff t_1 < t_2 \lor (t_1 = t_2 \land s_1 < s_2) \]

The impact of network delay on time is modeled through delay functions. Delay functions take a time and a client identifier and add the delay that the corresponding network connection adds.

\[ \text{delay}_{C \rightarrow S} : \text{Time} \times \text{Client} \rightarrow \text{Time}^* \]
\[ \forall c \in \text{Client}. \forall t, t' \in \text{Time}. (t < t') \implies \text{delay}_{C \rightarrow S}(t, c) < \text{delay}_{C \rightarrow S}(t', c) \]
\[ \forall c, c' \in \text{Client}. \forall t, t' \in \text{Time}. \text{delay}_{C \rightarrow S}(t, c) = (t', c') \implies t \leq t' \land c = c' \]
\[\text{delay}_{S \rightarrow C} : \text{Time}^s \times \text{Client} \rightarrow \text{Time}\]
\[
\forall c \in \text{Client}. \forall s, s' \in \text{Time}^s. (s < s') \implies \text{delay}_{S \rightarrow C}(s, c) < \text{delay}_{S \rightarrow C}(s', c)
\]
\[
\forall c \in \text{Client}. \forall (t, \text{slot}) \in \text{Time}^s. \text{delay}_{S \rightarrow C}((t, \text{slot}), c) > t
\]

\[
\text{Time}_{0,c}^s = \text{delay}_{C \rightarrow S}(\text{Time}_0, c)
\]
\[
\text{Time}_{\infty,c}^s = \text{delay}_{C \rightarrow S}(\text{Time}_{\infty}, c)
\]

\[
\forall c \in \text{Client}. (\text{Time}_0 < \text{Time}_{0,c} < \text{Time}_{\infty,c} < \text{Time}_{\infty})
\]
\[
\forall c \in \text{Client}. (\text{Time}_0 < \text{Time}_{0,c}^s < \text{Time}_{\infty,c}^s < \text{Time}_{\infty}^s)
\]

From here on we define the domains of the FRP primitives. We have events (sets of time-indexed values), behaviors (functions on time) and incremental behaviors (reified folds on events) on a client, session and application tier.

\[
[Event_{C}^S] = \left\{ e \in \text{Client} \rightarrow \mathcal{P}(\text{Time} \times [\tau]) \mid \begin{array}{c}
\text{finite}(e(c)) \\
\forall c \in \text{Client}. \left( \land \forall (t, v), (t', v') \in e(c). t = t' \implies v = v' \\
\land \forall (t, -) \in e(c). \text{Time}_{0,c}^s < t < \text{Time}_{\infty,c}^s \right) \end{array} \right\}
\]

\[
[Event_{\tau}^S] = \left\{ e \in \text{Client} \rightarrow \mathcal{P}(\text{Time}^s \times [\tau]) \mid \begin{array}{c}
\text{finite}(e(c)) \\
\forall c \in \text{Client}. \left( \land \forall (s, v), (s', v') \in e(c). s = s' \implies v = v' \\
\land \forall (s, -) \in e(c). \text{Time}_{0,c}^s < s < \text{Time}_{\infty,c}^s \right) \end{array} \right\}
\]
\[ \text{Event}^A = \left\{ \begin{array}{l} \left( e \in \mathcal{P}(\text{Time}^s \times \tau) \right) | \\
\text{finite}(e) \\
\land \forall(s, v), (s', v') \in e. s = s' \Rightarrow v = v' \\
\land \forall(s, -) \in e. \text{Time}^s_0 < s < \text{Time}^s_\infty \end{array} \right\} \]

\[ \text{Behavior}^C = \left\{ \begin{array}{l} b \in \text{Client} \rightarrow \text{Time} \rightarrow \tau | \\
\forall c \in \text{Client}. \text{dom}(b(c)) = \\
\left\{ t \in \text{Time} | \text{Time}^s_0, c \leq t \leq \text{Time}^s_\infty, c \right\} \end{array} \right\} \]

\[ \text{Behavior}^S = \left\{ \begin{array}{l} b \in \text{Client} \rightarrow \text{Time}^s \rightarrow \tau | \\
\forall c \in \text{Client}. \text{dom}(b(c)) = \left\{ s \in \text{Time}^s | \\
\text{Time}^s_0, c \leq s < \text{Time}^s_\infty, c \right\} \end{array} \right\} \]

\[ \text{Behavior}^A = \left\{ \begin{array}{l} b \in \text{Time}^s \rightarrow \tau | \\
\text{dom}(b) = \left\{ s \in \text{Time}^s | \\
\text{Time}^s_0 \leq s < \text{Time}^s_\infty \right\} \end{array} \right\} \]

\[ \text{IncBehavior}^C_{\tau, \delta} = \left\{ \begin{array}{l} (e, v, f) \in \\
[\text{Event}^C_{\delta}] \times (\text{Client} \rightarrow \tau) \times \text{Client} \rightarrow (\tau \times \delta \rightarrow \tau) \end{array} \right\} \]

\[ \text{IncBehavior}^S_{\tau, \delta} = \left\{ \begin{array}{l} (e, v, f) \in \\
[\text{Event}^S_{\delta}] \times (\text{Client} \rightarrow \tau) \times (\text{Client} \rightarrow (\tau \times \delta \rightarrow \tau)) \end{array} \right\} \]

\[ \text{IncBehavior}^A_{\tau, \delta} = \left\{ \begin{array}{l} (e, v_0, f) \in \\
[\text{Event}^A_{\delta}] \times \tau \times ((\tau \times \delta) \rightarrow \tau) \end{array} \right\} \]
Discrete behaviors are added for completeness but are special cases of incremental behaviors.

\[ \text{DiscBehavior}^C_{\tau} = \left\{ (e, v, f) \in \text{IncBehavior}^C_{\tau, \tau} \mid f = \lambda c. \lambda v. \lambda v'. v' \right\} \]

\[ \text{DiscBehavior}^S_{\tau} = \left\{ (e, v, f) \in \text{IncBehavior}^S_{\tau, \tau} \mid f = \lambda c. \lambda v. \lambda v'. v' \right\} \]

\[ \text{DiscBehavior}^A_{\tau} = \left\{ (e, v_0, f) \in \text{IncBehavior}^A_{\tau, \tau} \mid f = \lambda v. \lambda v'. v' \right\} \]

### A.2 Core Operations

Core operations and primitives are added to make it a usable language, e.g., composing FRP primitives as well as inspecting server status.

#### A.2.1 Client Information

Information regarding client identifiers and its connection status are added on the session and application tier.

**Definition 1.**

\[ \text{client} : \text{Behavior}^S_{\text{Client}} \]

\[ \left[ \text{client} \right] = \lambda c. \lambda s. \begin{cases} c & \text{if } Time^0_{0,c} \leq s < Time^*_{\infty,c} \\ \bot & \text{otherwise} \end{cases} \]

**Theorem 1.** \text{client} returns a valid result.

**Proof.** There is one property that the resulting value must comply with in order to be valid.

1. The domain is correct by definition.
Definition 2.

\[ \text{clientChanges} : \text{Event}^A_{\text{ClientChange}} \]

\[ [\text{clientChanges}] = \]

\[ \bigcup_{c \in \text{Client}} \{(\text{Time}_0^c, (\text{Connected}, c)), (\text{Time}_\infty^c, (\text{Disconnected}, c))\} \]

Theorem 2. \([\text{clientChanges}]\) returns a valid result.

Proof. There are 3 properties that the resulting value must comply with in order to be valid.

1. The result is finite since there is a finite number of clients.
2. All values are unique in (server)time since both \(\text{Time}_0^c\) and \(\text{Time}_\infty^c\) are indexed by \(c\) while \(\text{Time}_0^c < \text{Time}_\infty^c\) and are thus client-specific.
3. All values are within the required bounds since \(\text{Time}_0^c\) and \(\text{Time}_\infty^c\) is always between \(\text{Time}_0^c\) and \(\text{Time}_\infty^c\)

A.2.2 Events

The core operations \text{map}, \text{filter} and \text{union} are added to all tiers. All primitives are implemented similarly, the values in the event set of the input event(s) are inspected and either modified, dropped or merged to create a new set.

Definition 3.

\[ \text{map} : \text{Event}^C_\alpha \to (\alpha \to \beta) \to \text{Event}^C_\beta \]

\[ [\text{map}] (e, f) = \lambda c. \{ (t, f(v)) \mid (t, v) \in e(c) \} \]

Theorem 3. If the arguments are valid, \([\text{map}] (e, f)\) returns a valid result.

Proof. There are 3 properties that the resulting value must comply with in order to be valid.
1. No additional values are added so the result remains finite.

2. No changes are made to the time component of the event values, all relevant properties of \( e \) hold.

3. See above.

**Definition 4.**

\[
\text{map} : \text{Event}_\alpha^S \to (\alpha \to \beta) \to \text{Event}_\beta^S
\]

\[
[\text{map}] (e, f) = \lambda c. \{ (s, f(v)) \mid (s, v) \in e(c) \}
\]

**Theorem 4.** If the arguments are valid, \([\text{map}] (e, f) \) returns a valid result.

**Proof.** Similar to \( \text{Event}^C \)

**Definition 5.**

\[
\text{map} : \text{Event}_\alpha^A \to (\alpha \to \beta) \to \text{Event}_\beta^A
\]

\[
[\text{map}] (e, f) = \{ (s, f(v)) \mid (s, v) \in e \}
\]

**Theorem 5.** If the arguments are valid, \([\text{map}] (e, f) \) returns a valid result.

**Proof.** Similar to \( \text{Event}^C \)

**Definition 6.**

\[
\text{filter} : \text{Event}_\tau^C \to (\tau \to \text{Bool}) \to \text{Event}_\tau^C
\]

\[
[\text{filter}] (e, f) = \lambda c. \{ (t, v) \mid (t, v) \in e(c) \land f(v) \}
\]

**Theorem 6.** If the arguments are valid, \([\text{filter}] (e, f) \) returns a valid result.

**Proof.** There are 3 properties that the resulting value must comply with in order to be valid.

1. Event values are only being removed and not added so it remains finite.

2. No changes are made to the time component of the event values, all relevant properties of \( e \) hold.
3. See above.

Definition 7.

\[ \text{filter} : \text{Event}^S_{\tau} \rightarrow (\tau \rightarrow \text{Bool}) \rightarrow \text{Event}^S_{\tau} \]

\[ [\text{filter}] (e, f) = \lambda c. \{(s, v) \mid (s, v) \in e(c) \land f(v)\} \]

Theorem 7. If the arguments are valid, \([\text{filter}] (e, f)\) returns a valid result.

Proof. Similar to \(\text{Event}^C\)

Definition 8.

\[ \text{filter} : \text{Event}^A_{\tau} \rightarrow (\tau \rightarrow \text{Bool}) \rightarrow \text{Event}^A_{\tau} \]

\[ [\text{filter}] (e, f) = \{(s, v) \mid (s, v) \in e \land f(v)\} \]

Theorem 8. If the arguments are valid, \([\text{filter}] (e, f)\) returns a valid result.

Proof. Similar to \(\text{Event}^C\)

Definition 9.

\[ \text{union} : \text{Event}^C_{\tau} \rightarrow \text{Event}^C_{\tau} \rightarrow (\tau \rightarrow \tau \rightarrow \tau) \rightarrow \text{Event}^C_{\tau} \]

\[ [\text{union}] (e, e', f) = \]

\[ \lambda c. \text{let} \quad \text{left} = \{(t, v) \mid (t, v) \in e(c) \land \forall (t', -) \in e'(c). t \neq t'\} \]

\[ \text{both} = \{(t, f(v, v')) \mid (t, v) \in e(c) \land (t', v') \in e'(c) \land t = t'\} \]

\[ \text{right} = \{(t, v) \mid (t, v) \in e'(c) \land \forall (t', -) \in e(c). t \neq t'\} \]

in left \cup both \cup right

Theorem 9. If the arguments are valid, \([\text{union}] (e, e', f)\) returns a valid result.

Proof. There are 3 properties that the resulting value must comply with in order to be valid.

1. The union of three finite sets remains finite.
2. The event values are unique in time by construction, *both* handles exactly this case, merging two values into one with *f* whenever the event values have the same time component.

3. The event values are within the correct bounds since both *e* and *e*′ are too.

**Definition 10.**

\[
\text{union} : \text{Event}_\tau^S \rightarrow \text{Event}_\tau^S \rightarrow (\tau \rightarrow \tau \rightarrow \tau) \rightarrow \text{Event}_\tau^S
\]

\[
\begin{align*}
\text{union}(e, e', f) &= \\
&= \lambda c. \text{let} \quad \text{left} = \{(s, v) \mid (s, v) \in e(c) \land \forall(s', -) \in e'(c). s \neq s'\} \\
&\quad \text{both} = \{(s, f(v, v')) \mid (s, v) \in e(c) \land (s', v') \in e'(c) \land s = s'\} \\
&\quad \text{right} = \{(s, v) \mid (s, v) \in e'(c) \land \forall(s', -) \in e(c). s \neq s'\} \\
&\quad \text{in left} \cup \text{both} \cup \text{right}
\end{align*}
\]

**Theorem 10.** If the arguments are valid, \(\text{union}(e, e', f)\) returns a valid result.

**Proof.** Similar to Event^C

**Definition 11.**

\[
\text{union} : \text{Event}_\tau^A \rightarrow \text{Event}_\tau^A \rightarrow (\tau \rightarrow \tau \rightarrow \tau) \rightarrow \text{Event}_\tau^A
\]

\[
\begin{align*}
\text{union}(e, e', f) &= \\
&= \text{let} \quad \text{left} = \{(s, v) \mid (s, v) \in e \land \forall(s', -) \in e'. s \neq s'\} \\
&\quad \text{both} = \{(s, f(v, v')) \mid (s, v) \in e \land (s', v') \in e' \land s = s'\} \\
&\quad \text{right} = \{(s, v) \mid (s, v) \in e' \land \forall(s', -) \in e. s \neq s'\} \\
&\quad \text{in left} \cup \text{both} \cup \text{right}
\end{align*}
\]

**Theorem 11.** If the arguments are valid, \(\text{union}(e, e', f)\) returns a valid result.

**Proof.** Similar to Event^C
A.2.3 Behavior

Constant behaviors as well as their core primitive map2 are added on all tiers. All operations are implemented by either producing a constant function (constant) or by creating a function which in turn polls the input behaviors at the right time (map2).

Definition 12.

\[ \text{constant} : \alpha \rightarrow \text{Behavior}^C_{\alpha} \]

\[ \llbracket \text{constant} \rrbracket(a) = \lambda c. \lambda t. \begin{cases} a & \text{if } \text{Time}_0 < t < \text{Time}_\infty, \\ \perp & \text{otherwise} \end{cases} \]

Theorem 12. If the arguments are valid, \( \llbracket \text{constant} \rrbracket(a) \) returns a valid result.

Proof. There is one property that the resulting value must comply with in order to be valid.

1. The domain is correct by definition.

Definition 13.

\[ \text{constant} : \alpha \rightarrow \text{Behavior}^S_{\alpha} \]

\[ \llbracket \text{constant} \rrbracket(a) = \lambda c. \lambda s. \begin{cases} a & \text{if } \text{Time}_0^s < s < \text{Time}_\infty^s, \\ \perp & \text{otherwise} \end{cases} \]

Theorem 13. If the arguments are valid, \( \llbracket \text{constant} \rrbracket(a) \) returns a valid result.

Proof. Similar to Event^C.

Definition 14.

\[ \text{constant} : \alpha \rightarrow \text{Behavior}^A_{\alpha} \]

\[ \llbracket \text{constant} \rrbracket(a) = \lambda s. \begin{cases} a & \text{if } \text{Time}_0^s \leq s < \text{Time}_\infty^s, \\ \perp & \text{otherwise} \end{cases} \]

Theorem 14. If the arguments are valid, \( \llbracket \text{constant} \rrbracket(a) \) returns a valid result.
Proof. Similar to Event\(^C\)  

Definition 15.

\[ \text{map2} : \text{Behavior}_\alpha^C \rightarrow \text{Behavior}_\beta^C \rightarrow (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \text{Behavior}_\gamma^C \]

\[ \llbracket \text{map2} \rrbracket (b, b', f) = \lambda c. \lambda t. \begin{cases} f(b(c)(t), b'(c)(t)) \quad & \text{if } \text{Time}_{\alpha,c} \leq t < \text{Time}_{\infty,c} \\ \bot \quad & \text{otherwise} \end{cases} \]

Theorem 15. If the arguments are valid, \( \llbracket \text{map2} \rrbracket (b, b', f) \) returns a valid result.

Proof. There is one property that the resulting value must comply with in order to be valid.

1. The domain is valid by construction.

Definition 16.

\[ \text{map2} : \text{Behavior}_\alpha^S \rightarrow \text{Behavior}_\beta^S \rightarrow (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \text{Behavior}_\gamma^S \]

\[ \llbracket \text{map2} \rrbracket (b, b', f) = \lambda c. \lambda s. \begin{cases} f(b(c)(s), b'(c)(s)) \quad & \text{if } \text{Time}_{\alpha,c}^s \leq s < \text{Time}_{\infty,c}^s \\ \bot \quad & \text{otherwise} \end{cases} \]

Theorem 16. If the arguments are valid, \( \llbracket \text{map2} \rrbracket (b, b', f) \) returns a valid result.

Proof. Similar to Event\(^C\)  

Definition 17.

\[ \text{map2} : \text{Behavior}_\alpha^A \rightarrow \text{Behavior}_\beta^A \rightarrow (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \text{Behavior}_\gamma^A \]

\[ \llbracket \text{map2} \rrbracket (b, b', f) = \lambda s. \begin{cases} f(b(s), b'(s)) \quad & \text{if } \text{Time}_{0}^s \leq s < \text{Time}_{\infty}^s \\ \bot \quad & \text{otherwise} \end{cases} \]

Theorem 17. If the arguments are valid, \( \llbracket \text{map2} \rrbracket (b, b', f) \) returns a valid result.

Proof. Similar to Event\(^C\)
A.2.4 Incremental Behavior

Incremental behaviors can be constant and can be composed through \( \text{map2} \). Constant behaviors have the empty set as its deltas and an initial value with a placeholder identity functions as a fold. \( \text{map2} \) is implemented on each tier in the same steps:

- Turn the input incremental behaviors into behaviors to easily retrieve the “current value”.
- Combine the input behaviors.
- Compose the input incremental behaviors’ deltas by making a distinction between the cases where either both behaviors change or one of them changes.
- Compose the new delta event by using the previously composed behaviors and deltas to derive the new event.
- Finalize the result by filtering out all empty values from its delta stream and by composing the initial values of the inputs.

**Definition 18.**

\[
\text{constant} : \alpha \rightarrow \text{IncBehavior}^C_{\alpha, \delta}
\]

\[
[\text{constant}] (a) = (\lambda c. \emptyset, \lambda c. a, \lambda c. \lambda v. \lambda d. v)
\]

**Theorem 18.** If the arguments are valid, \([\text{constant}] (a) \) returns a valid result.

**Proof.** There is one property that the resulting value must comply with in order to be valid.

1. The 3 constructed elements of the tuple are of the correct type.

**Definition 19.**

\[
\text{constant} : \alpha \rightarrow \text{IncBehavior}^S_{\alpha, \delta}
\]

\[
[\text{constant}] (a) = (\lambda c. \emptyset, \lambda c. a, \lambda c. \lambda v. \lambda d. v)
\]

**Theorem 19.** If the arguments are valid, \([\text{constant}] (a) \) returns a valid result.

**Proof.** Similar to \( \text{Event}^C \)
Definition 20.

\[ \text{constant} : \alpha \to \text{IncBehavior}^{A}_{\alpha, \delta} \]

\[ \text{constant}(a) = (\emptyset, a, \lambda v. \lambda d. v) \]

Theorem 20. If the arguments are valid, \([\text{constant}] (a)\) returns a valid result.

Proof. Similar to Event\textsuperscript{C} \(\Box\)

Definition 21.

\[ [\text{Inc}_{\delta, \delta'}] = \{ \text{Left}(x) \mid x \in [\delta] \} \cup \{ \text{Right}(y) \mid y \in [\delta'] \} \cup \{ \text{All}(x, y) \mid x \in [\delta], y \in [\delta'] \} \]

\[ \text{incMap2} : \text{IncBehavior}^{C}_{\tau, \delta} \to \text{IncBehavior}^{C}_{\tau', \delta'} \to (\tau \to \tau' \to \tau'') \]

\[ \to (\tau \to \tau' \to \text{Inc}_{\delta, \delta'} \to \delta'') \to (\tau'' \to \delta'' \to \tau'') \to \text{IncBehavior}^{C}_{\tau'', \delta''} \]

\[ [\text{incMap2}]((e, v, f), (e', v', f'), fi, fd, ff) = \]

let \( b = [\text{convert}]((e, v, f)) \)

\( b' = [\text{convert}]((e', v', f')) \)

\( bb = [\text{map}]2(b, b', \lambda x. \lambda y. (x, y)) \)

\( de = [\text{map}](e, \lambda x. \text{Left}(x)) \)

\( de' = [\text{map}](e', \lambda x. \text{Right}(x)) \)

\( de'' = [\text{union}](de, de', \lambda (\text{Left}(x)). \lambda (\text{Right}(y)). \text{All}(x, y)) \)

\( e'' = [\text{snapshot}](bb, [\text{map}](de'', \lambda \text{dev}. \lambda (bv, bv'). \text{fd}(bv)(bv')(\text{dev}))) \)

\( \text{in}([\text{filter}](e'', \lambda x. x \neq \bot), \lambda c. \text{fi}(v(c), v'(c)), ff) \)

Theorem 21. If the arguments are valid, \([\text{incMap2}]((e, v, f), (e', v', f'), fi, fd, ff)\) returns a valid result.

Proof. There is one property that the resulting value must comply with in order to be valid.
1. The 3 constructed elements of the tuple are of the correct type.

Definition 22.

\[
incMap2 : IncBehavior^{S, \delta}_\tau \rightarrow IncBehavior^{S, \delta'}_\tau' \rightarrow (\tau \rightarrow \tau' \rightarrow \tau'') \\
\rightarrow (\tau \rightarrow \tau' \rightarrow Inc_{\delta, \delta'} \rightarrow \delta'') \rightarrow (\tau'' \rightarrow \delta'' \rightarrow \tau'') \rightarrow IncBehavior^{S, \delta''}_\tau''
\]

\[
\lbrack incMap2 \rbrack((e, v, f), (e', v', f'), fi, fd, ff) =
\]

\[
\begin{align*}
\text{let } & b = \lbrack convert \rbrack((e, v, f)) \\
& b' = \lbrack convert \rbrack((e', v', f')) \\
& bb = \lbrack map2 \rbrack(b, b', \lambda x. \lambda y. (x, y)) \\
& de = \lbrack map \rbrack(e, \lambda x. \text{Left}(x)) \\
& de' = \lbrack map \rbrack(e', \lambda x. \text{Right}(x)) \\
& de'' = \lbrack union \rbrack(de, de', \lambda (\text{Left}(x)). \lambda (\text{Right}(y)). \text{All}(x, y)) \\
& e'' = \lbrack snapshot \rbrack(bb, \lbrack map \rbrack(de'', \lambda dev. \lambda (bv, bv'). fd(bv)(bv')(dev))) \\
& \text{in(\lbrack filter \rbrack(e'', \lambda x. x \neq \bot), \lambda c. fi(v(c), v'(c)), ff)}
\end{align*}
\]

Theorem 22. If the arguments are valid, \lbrack incMap2 \rbrack((e, v, f), (e', v', f'), fi, fd, ff) returns a valid result.

Proof. Similar to Event\textsuperscript{C}.

Definition 23.

\[
incMap2 : IncBehavior^{A, \delta}_\tau \rightarrow IncBehavior^{A, \delta'}_\tau' \rightarrow (\tau \rightarrow \tau' \rightarrow \tau'') \\
\rightarrow (\tau \rightarrow \tau' \rightarrow Inc_{\delta, \delta'} \rightarrow \delta'') \rightarrow (\tau'' \rightarrow \delta'' \rightarrow \tau'') \rightarrow IncBehavior^{A, \delta''}_\tau''
\]
\[
\text{let } b &= \text{convert}((e, v, f)) \\
b' &= \text{convert}((e', v', f')) \\
bb &= \text{map2}(b, b', \lambda x. \lambda y. (x, y)) \\
de &= \text{map}(e, \lambda x. \text{Left}(x)) \\
de' &= \text{map}(e', \lambda x. \text{Right}(x)) \\
de'' &= \text{union}(de, de', \lambda (\text{Left}(x)). \lambda (\text{Right}(y)). \lambda (x, y)) \\
e'' &= \text{snapshot}(bb, \text{map}(de'', \lambda \text{dev}. \lambda (\text{bv}, \text{bv'}). \text{fd}(\text{bv})(\text{bv'})(\text{dev}))) \\
\text{in}(\text{filter}(e'', \lambda x. x \neq \bot), fi(v, v'), ff)
\]

**Theorem 23.** If the arguments are valid, \([\text{incMap2}((e, v, f), (e', v', f'), fi, fd, ff)]\) returns a valid result.

*Proof.* Similar to Event$^C$ \qed

### A.2.5 Discrete Behavior

We can define discrete map two in terms of the incremental version since we defined earlier that semantically discrete behaviors are a subset of incremental behaviors.

\[
discMap2 : \text{DiscBehavior}^C_\alpha \to \text{DiscBehavior}^C_\beta \to (\alpha \to \beta \to \tau) \to \text{DiscBehavior}^C_\tau \\
\text{let } d &= \text{convert}(d) \\
\text{let } b &= \text{convert}(b) \\
b' &= \text{convert}(b') \\
\text{let } c &= \text{convert}(c) \\
\text{let } v &= \text{convert}(v) \\
v' &= \text{convert}(v') \\
\text{let } x &= \text{convert}(x) \\
x' &= \text{convert}(x') \\
\text{let } y &= \text{convert}(y) \\
y' &= \text{convert}(y') \\
\text{let } f &= \text{convert}(f) \\
\text{let } f' &= \text{convert}(f') \\
\text{let } fi &= \text{convert}(fi) \\
\text{let } fi' &= \text{convert}(fi') \\
\text{let } fd &= \text{convert}(fd) \\
\text{let } fd' &= \text{convert}(fd') \\
\text{let } ff &= \text{convert}(ff) \\
\text{let } ff' &= \text{convert}(ff') \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert}(\lambda) \\
\text{let } \lambda &= \text{convert(}
\[ \text{discMap2} : \text{DiscBehavior}_\alpha^S \rightarrow \text{DiscBehavior}_\beta^S \rightarrow (\alpha \rightarrow \beta \rightarrow \tau) \rightarrow \text{DiscBehavior}_\tau^S \]

\[
\left[ \text{discMap2} \right](b, b', f) =
\]

\[
\left[ \text{incMap2} \right](b, b', f, x, y, d, \lambda, \lambda', \lambda', v, v') =
\]

\[ \text{discMap2} : \text{DiscBehavior}_\alpha^A \rightarrow \text{DiscBehavior}_\beta^A \rightarrow (\alpha \rightarrow \beta \rightarrow \tau) \rightarrow \text{DiscBehavior}_\tau^A \]

\[
\left[ \text{discMap2} \right](b, b', f) =
\]

\[
\left[ \text{incMap2} \right](b, b', f, x, y, d, \lambda, \lambda', \lambda', v, v') =
\]

### A.3 Conversion Primitives

In this section we add conversion primitives. We add session to application and vice versa tier-crossing primitives. Since these reside on the same physical tier (server) they do not cross the network and no delays are added.

The second type of conversions that are added are incremental behavior to behavior conversions. These execute the reified fold of the incremental behavior to produce a function of time that can be used as a regular behavior.

Note that bootstrapping is defined in all primitives that go from the application to the session tier. The value at the start of the session tier later determines the value at the start of the client tier which in turn provisions the client tier with the proper initial value, i.e., solving the bootstrapping problem.
A.3.1 Events

Definition 24.

\[
\text{toApplication} : \text{Event}^S_\tau \rightarrow \text{Event}^A_{\text{Client} \rightarrow \tau}
\]

\[
\begin{align*}
\text{toApplication}(e) = \\
\text{let allOccurrences} = \bigcup_{c \in \text{Client}} \{(s, (c, v)) \mid (s, v) \in e(c)\} \\
\text{in } \bigcup_{s \in \text{Time}^s} \text{Time}^s_0 < s < \text{Time}^s_{\infty} \\
\left\{(s, set) \mid set' = \{(c, v) \mid (s', (c, v)) \in \text{allOccurrences} \land s = s'\} \land \\
set' \neq \emptyset \land \text{set} = \text{fromSet}(set') \right\}
\end{align*}
\]

\[
\text{fromSet} : \mathcal{P}((\text{Client} \times \tau) \rightarrow \text{Client} \rightarrow \tau)
\]

\[
\text{fromSet}(s) = \lambda c. \left\{ \begin{array}{ll}
\text{if } (v, c) \in s \\
\bot & \text{otherwise}
\end{array} \right.
\]

Theorem 24. If the arguments are valid, \([\text{toApplication}](e)\) returns a valid result.

Proof. There are 3 properties that the resulting value must comply with in order to be valid.

1. There is a finite number of clients which implies that a union of all values for all connected clients is finite (a union of finite amount of finite sets).

2. We group all occurrences per time slot in a partial function using fromSet, as such the resulting set is unique in time since the original occurrences were unique in time per client.

3. Event$^S$s have time bounds that are always within the bounds of Event$^A$s and we only take sets that contain values from Event$^S$s.

\qed

Definition 25.

\[
\text{toSession} : \text{Event}^A_\tau \rightarrow \text{Event}^S_\tau
\]

\[
\begin{align*}
\text{toSession}(e) = \lambda c. \{(s, v) \mid (s, v) \in e \land \text{Time}^*_0, c < s < \text{Time}^*_\infty, c\}
\end{align*}
\]
Theorem 25. If the arguments are valid, \( \llbracket \text{toSession} \rrbracket (e) \) returns a valid result.

Proof. There are 3 properties that the resulting value must comply with in order to be valid.

1. The set of all events for a client is at most the same size and distribution as the set of all events for all clients. The latter is already required to be finite by \( \text{Event}^A \)‘s.
2. All event values are unique in time for \( \text{Event}^A \)‘s and per event value only one value is calculated per client so this is obvious.
3. We only select occurrences that fit the required time bounds.

A.3.2 Behavior

Definition 26.

\( \text{toApplication} : \text{Behavior}_{\tau}^S \rightarrow \text{Behavior}_{\text{Client} \rightarrow \tau}^A \)

\[ \llbracket \text{toApplication} \rrbracket (b) = \lambda s . \begin{cases} \lambda c \rightarrow b(c)(s) & \text{if } \text{Time}^s_0 \leq s < \text{Time}^s_{\infty} \\ \bot & \text{otherwise} \end{cases} \]

Theorem 26. If the arguments are valid, \( \llbracket \text{toApplication} \rrbracket (b) \) returns a valid result.

Proof. There is one property that the resulting value must comply with in order to be valid.

1. The domain of the resulting function is within the required bounds by construction.

Definition 27.

\( \text{toSession} : \text{Behavior}_{\tau}^A \rightarrow \text{Behavior}_{\tau}^S \)

\[ \llbracket \text{toSession} \rrbracket (b) = \lambda c . \lambda s . \begin{cases} b(s) & \text{if } \text{Time}^s_{0,c} \leq s < \text{Time}^s_{\infty,c} \\ \bot & \text{otherwise} \end{cases} \]
Theorem 27. If the arguments are valid, \([\text{toSession}] (b)\) returns a valid result.

Proof. There is one property that the resulting value must comply with in order to be valid.

1. The domain of the resulting function is within the required bounds by construction.

\[\blacksquare\]

A.3.3 Incremental Behavior

Definition 28.

\[\text{toApplication} : \text{IncBehavior}^S_{\tau, \delta} \rightarrow \text{IncBehavior}^A_{\text{Client} \rightarrow \tau, \text{Client} \rightarrow \delta \times (\text{ClientChanges} \sqcup \top)}\]

\([\text{toApplication}] ( (e, v_0, f) ) = \]

\[
\text{let } e' = [\text{map}][[\text{toApplication}](e), \lambda \text{map}. (\text{map}, \top)]
\]

\[
\text{clientChanges}' = [\text{map}][[\text{clientChanges}], \lambda \text{c}. (\lambda _, \bot, c)]
\]

\[
u = [\text{union}](e', \text{clientChanges}', \lambda \text{c}. (\text{map}_1, c_2))
\]

\[
f' = \lambda v. \lambda d. \lambda c. \left\{\begin{array}{ll}
v_0(c) & \text{if } d.2 = (\text{Connected}, c) \\
\bot & \text{if } d.2 = (\text{Disconnected}, c) \\
v(c) & \text{if } d.1(c) = \bot \\
f(c)(v(c))(d.1(c)) & \text{otherwise}
\end{array}\right.
\]

\[
\text{in}(u, \lambda c \rightarrow \bot, f')
\]

Theorem 28. If the arguments are valid, \([\text{toApplication}] (b)\) returns a valid result.

Proof. There is one property that the resulting value must comply with in order to be valid.

1. The 3 constructed elements of the resulting tuple are of the correct type.

\[\blacksquare\]
Definition 29.

\[ \text{toSession} : \text{IncBehavior}^A_{\tau,\delta} \rightarrow \text{IncBehavior}^S_{\tau,\delta} \]

\[ \llbracket \text{toSession} \rrbracket ((e, v, f)) = (\llbracket \text{toSession} \rrbracket (e), \lambda c. \llbracket \text{convert} \rrbracket ((e, v, f))(\text{Time}^*_c), \lambda c. f) \]

Theorem 29. If the arguments are valid, \( \llbracket \text{toSession} \rrbracket ((e, v, f)) \) returns a valid result.

Proof. There is one property that the resulting value must comply with in order to be valid.

1. The 3 constructed elements of the resulting tuple are of the correct type.

\[ \Box \]

A.3.4 Events ↔ Behaviors

Definition 30.

\[ \text{snapshot} : \text{Event}^C_{\alpha\rightarrow\beta} \rightarrow \text{Behavior}^C_{\alpha} \rightarrow \text{Event}^C_{\beta} \]

\[ \llbracket \text{snapshot} \rrbracket (e, b) = \lambda c. \{(t, v(b(c)(t))) \mid (t, v) \in e(c)\} \]

Theorem 30. If the arguments are valid, \( \llbracket \text{snapshot} \rrbracket (e, b) \) returns a valid result.

Proof. There are 3 properties that the resulting value must comply with in order to be valid.

1. A value is created for each value in \( e \) which is a finite amount.
2. Only one value is created for every value in \( e \) so the result remains unique in time. Note that the time bounds on \( e \) ensure that \( b(c)(t) \) is well defined.
3. The bounds do not change compared to \( e \).

\[ \Box \]

Definition 31.

\[ \text{snapshot} : \text{Event}^S_{\alpha\rightarrow\beta} \rightarrow \text{Behavior}^S_{\alpha} \rightarrow \text{Event}^S_{\beta} \]

\[ \llbracket \text{snapshot} \rrbracket (e, b) = \lambda c. \{(s, v(b(c)(s))) \mid (s, v) \in e(c)\} \]
**Theorem 31.** If the arguments are valid, \( \langle \text{snapshot} \rangle (e, b) \) returns a valid result.

**Proof.** Similar to Event^C. □

**Definition 32.**

\[
\text{snapshot} : \text{Event}^A_{\alpha \rightarrow \beta} \rightarrow \text{Behavior}^A_{\alpha} \rightarrow \text{Event}^A_{\beta}
\]

\[
\langle \text{snapshot} \rangle (e, b) = \{(s, v(b(s))) \mid (s, v) \in e\}
\]

**Theorem 32.** If the arguments are valid, \( \langle \text{snapshot} \rangle (e, b) \) returns a valid result.

**Proof.** Similar to Event^C. □

**Definition 33.**

\[
\text{foldP} : \text{Event}^S_{\delta} \rightarrow \tau \rightarrow (\tau \rightarrow \delta \rightarrow \tau) \rightarrow \text{IncBehavior}^C_{\tau, \delta}
\]

\[
\langle \text{foldP} \rangle (e, v, f) = (e, \lambda c. v, \lambda c. f)
\]

**Theorem 33.** If the arguments are valid, \( \langle \text{foldP} \rangle (e, v, f) \) returns a valid result.

**Proof.** There is one property that the resulting value must comply with in order to be valid.

1. The 3 constructed elements of the resulting tuple are of the correct type. □

**Definition 34.**

\[
\text{foldP} : \text{Event}^S_{\delta} \rightarrow \tau \rightarrow (\tau \rightarrow \delta \rightarrow \tau) \rightarrow \text{IncBehavior}^C_{\tau, \delta}
\]

\[
\langle \text{foldP} \rangle (e, v, f) = (e, \lambda c. v, \lambda c. f)
\]

**Theorem 34.** If the arguments are valid, \( \langle \text{foldP} \rangle (e, v, f) \) returns a valid result.

**Proof.** Similar to Event^C. □

**Definition 35.**

\[
\text{foldP} : \text{Event}^A_{\delta} \rightarrow \tau \rightarrow (\tau \rightarrow \delta \rightarrow \tau) \rightarrow \text{IncBehavior}^A_{\tau, \delta}
\]

\[
\langle \text{foldP} \rangle (e, v, f) = (e, v, f)
\]

**Theorem 35.** If the arguments are valid, \( \langle \text{foldP} \rangle (e, v, f) \) returns a valid result.

**Proof.** Similar to Event^C. □
A.3.5 Incremental Behavior → Behavior

**Definition 36.**

\[
\text{convert} : \text{IncBehavior}_{\tau,\delta}^{C} \rightarrow \text{Behavior}_{\tau}^{C}
\]

\[\llbracket \text{convert} \rrbracket ( (e, v, f)) = \]

\[
\lambda c. \lambda s. \begin{cases} 
  f(c)(f(c)(...f(c)(v(c), d_1), ...), d_{n-1})d_n) & \text{if } \text{Time}_{0,c} \leq t < \text{Time}_{\infty,c} \wedge \\
  e(c) = \{(t_1, d_1), ..., (t_n, d_n), (t_{n+1}, d_{n+1}), ...\} \wedge \\
  t_1 < ... < t_n \leq t < t_{n+1} < ... & \text{otherwise} \\
\end{cases}
\]

**Theorem 36.** If the arguments are valid, \(\llbracket \text{convert} \rrbracket ( (e, v, f))\) returns a valid result.

**Proof.** There is one property that the resulting value must comply with in order to be valid.

1. The domain of the resulting function is within the required bounds by construction and it is well defined since an order can be given to the timestamps of \(e(c)\) due to time being ordered and the amount of event values being finite.

**Definition 37.**

\[
\text{convert} : \text{IncBehavior}_{\tau,\delta}^{S} \rightarrow \text{Behavior}_{\tau}^{S}
\]

\[\llbracket \text{convert} \rrbracket ( (e, v, f)) = \]

\[
\lambda c. \lambda s. \begin{cases} 
  f(c)(f(c)(...f(c)(v(c), d_1), ...), d_{n-1})d_n) & \text{if } \text{Time}^{s,c}_{0} \leq s < \text{Time}^{s,c}_{\infty} \wedge \\
  e(c) = \{(s_1, d_1), ..., (s_n, d_n), (s_{n+1}, d_{n+1}), ...\} \wedge \\
  s_1 < ... < s_n \leq s < s_{n+1} < ... & \text{otherwise} \\
\end{cases}
\]

**Theorem 37.** If the arguments are valid, \(\llbracket \text{convert} \rrbracket ( (e, v, f))\) returns a valid result.

**Proof.** Similar to \(\text{Event}^{C}\)
Definition 38.

\[
\text{convert} : \text{IncBehavior}_{\tau, \delta}^A \rightarrow \text{Behavior}_{\tau}^A
\]

\[
\llbracket \text{convert} \rrbracket((e, v, f)) =
\begin{cases}
  f(f(...(f(v, d_1), ...), d_{n-1})d_n) & \text{if } \text{Time}^e_0 \leq s < \text{Time}^s_{\infty} \land \varepsilon = \{(s_1, d_1), ..., (s_n, d_n), (s_{n+1}, d_{n+1}), \ldots\} \land \\
  \lambda s. \quad s_1 \ldots \leq s < s_{n+1} < ... & \text{otherwise}
\end{cases}
\]

Theorem 38. If the arguments are valid, \(\llbracket \text{convert} \rrbracket((e, v, f))\) returns a valid result.

Proof. Similar to Event\(^C\).

A.4 Tier-Crossing Primitives

The following primitives turn the language from essentially two (Application + Session / Client) separate FRP languages into a multi-tier FRP language suitable to program client/server web applications. These conversions cross a network and add delays to time components.

A.4.1 Events

Definition 39.

\[
\text{toServer} : \text{Event}_{\tau}^C \rightarrow \text{Event}_{\tau}^S
\]

\[
\llbracket \text{toServer} \rrbracket(e) = \lambda c. \{(\text{delay}_{C \rightarrow S}(t, c), v) \mid (t, v) \in e(c)\}
\]

Theorem 39. If the arguments are valid, toServer(e) returns a valid result.

Proof. There are 3 properties that the resulting value must comply with in order to be valid.

1. Event\(^C\) is already finite, we do not add any elements.

2. The \(\text{delay}_{C \rightarrow S}\) function is injective since it is a strictly monotone function. Since Event\(^C\)'s are unique in time, the resulting Event\(^S\) will remain unique in time after applying \(\text{delay}_{C \rightarrow S}\).
3. Client events are bound for each client by $\text{Time}_{0,c}$ and $\text{Time}_{\infty,c}$, since all events are delayed equally per client with $\text{delay}_{C \rightarrow S}$ they are bounded by $\text{Time}^\delta_{0,c}$ and $\text{Time}^\delta_{\infty,c}$.

\[\square\]

**Definition 40.**

$$\text{toClient} : \text{Event}^S_T \rightarrow \text{Event}^C_T$$

$$\llbracket \text{toClient} \rrbracket (e) = \lambda c. \begin{cases} (s, v) \in e(c) \\ (t, v) \land t = \text{delay}_{S \rightarrow C}(s, c) \\ \land t < \text{Time}_{\infty,c} \end{cases}$$

**Theorem 40.** If the arguments are valid, $\llbracket \text{toClient} \rrbracket (e)$ returns a valid result.

**Proof.** There are 3 properties that the resulting value must comply with in order to be valid.

1. $\text{Event}^S$ is already finite, we do not add any elements.

2. The $\text{delay}_{S \rightarrow C}$ function is injective since it is a strictly monotone function. Since $\text{Event}^C$'s are unique in time, the resulting $\text{Event}^S$ will remain unique in time after applying $\text{delay}_{S \rightarrow C}$.

3. Events in $\text{Event}^S$'s are bounded by $\text{Time}^\delta_{0,c}$ and $\text{Time}^\delta_{\infty,c}$. $\text{delay}_{S \rightarrow C}(\text{Time}^\delta_{0,c}, c)$ is by definition larger than $\text{Time}_{0,c}$ (the required lower bound) because delays are increasing.

The upper bound holds by definition of $\text{toClient}$.

\[\square\]

**A.4.2 Incremental Behavior**

**Definition 41.**

$$\text{toServer} : \text{IncBehavior}^C_{\tau,\delta} \rightarrow \text{IncBehavior}^S_{\tau,\delta}$$

$$\llbracket \text{toServer} \rrbracket (b) = \text{let}(e, v, f) = b$$

$$\text{in} (\llbracket \text{toServer} \rrbracket (e), v, f)$$
**Theorem 41.** If the arguments are valid, \( \llbracket \text{toServer} \rrbracket (b) \) returns a valid result.

**Proof.** There is one property that the resulting value must comply with in order to be valid.

1. The 3 constructed elements of the resulting tuple are of the correct type.

\[ \square \]

**Definition 42.**

\[
\text{toClient} : \text{IncBehavior}_{\tau,\delta}^S \rightarrow \text{IncBehavior}_{\tau,\delta}^C
\]

\[
\llbracket \text{toClient} \rrbracket (i) = \text{let} (e, v, f) = i
\]

\[
\text{in} (\llbracket \text{toClient} \rrbracket (e), v, f)
\]

**Theorem 42.** If the arguments are valid, \( \llbracket \text{toClient} \rrbracket (i) \) returns a valid result.

**Proof.** There is one property that the resulting value must comply with in order to be valid.

1. The 3 constructed elements of the resulting tuple are of the correct type.

\[ \square \]

### A.5 Commutative Tier-Crossing

An example property that can be proven with these denotational semantics are these instances of commutative tier-crossing where e.g., first mapping over an event and then crossing the network is functionally the same as first crossing the network and then mapping the event.

Because of tiered glitch freedom, the same property holds even in the case of a union where two events are required. It is functionally the same to bring a merged client event to the session tier as it is to first bring both client events to the session tier and merge them afterwards.

**Definition 43.**

\[
\llbracket \text{toServer} \rrbracket (\llbracket \text{map} \rrbracket (e,f)) = \llbracket \text{map} \rrbracket (\llbracket \text{toServer} \rrbracket (e), f)
\]
Proof.

$$[\text{toServer}](\lambda c. \{(t, f(v)) \mid (t, v) \in e(c)\}) = [\text{map}]$$

$$\lambda c. \{(\text{delay}_{C \rightarrow S}(t, c), v) \mid (t, v) \in (\lambda c. \{(t, f(v)) \mid (t, v) \in e(c)\})(c)\} = [\text{toServer}]$$

$$\lambda c. \{(t, f(v), v) \mid (t, v) \in (\lambda c. \{(\text{delay}_{C \rightarrow S}(t, c)) \mid (t, v) \in e(c)\})(c)\} = \text{rewrite}$$

$$\lambda c. \{(t, f(v), v) \mid (t, v) \in ([\text{toServer}]_e(c))((c)\} = [\text{toServer}]^{-1}$$

$$[\text{map}](\text{toServer}(e, f)) = [\text{map}]^{-1}$$

\[\Box\]

Definition 44.

$$[\text{toServer}](\text{union}(e, e', f)) = \text{union}([\text{toServer}](e), [\text{toServer}](e'), f)$$

Proof.

$$\lambda c. \text{let left } = \left\{ (t, v) \mid \begin{array}{l}
(t, v) \in e(c) \\
\land \forall (t', -) \in e'(c). t \neq t'
\end{array} \right\} = [\text{union}]$$

$$\begin{array}{l}
\text{both } = \left\{ (t, f(v, v')) \mid \\
(t, v) \in e(c) \\
\land (t', v') \in e'(c) \\
\land t = t'
\end{array} \right\}$$

$$\begin{array}{l}
\text{right } = \left\{ (t, v) \mid \\
(t, v) \in e'(c) \\
\land \forall (t', -) \in e(c). t \neq t'
\end{array} \right\}$$

$$\text{in}[\text{toServer}](\text{left} \cup \text{both} \cup \text{right})$$
\[ \lambda c. \textbf{let } \text{left} = \left\{ \begin{array}{l}
  (\text{delay}_{C \rightarrow S}(t, c), v) \\
  (t, v) \in e(c) \\
  \land \forall (t', -) \in e'(c). \ t \neq t'
\end{array} \right\} = \text{[toServer]} \]

\[ \text{both} = \left\{ \begin{array}{l}
  (\text{delay}_{C \rightarrow S}(t, c), f(v, v')) \\
  (t, v) \in e(c) \\
  \land (t', v') \in e'(c) \\
  \land t = t'
\end{array} \right\} \]

\[ \text{right} = \left\{ \begin{array}{l}
  (\text{delay}_{C \rightarrow S}(t, c), v) \\
  (t, v) \in e'(c) \\
  \land \forall (t', -) \in e(c). \ t \neq t'
\end{array} \right\} \]

\text{in left} \cup \text{both} \cup \text{right}
\[
\begin{align*}
\lambda c. \text{let} & \quad eS = \lambda c. \{(\text{delay}_{C \rightarrow S}(t, c), v) \mid (t, v) \in e(c)\} \quad \text{rewrite} \\
& \quad eS' = \lambda c. \{(\text{delay}_{C \rightarrow S}(t, c), v) \mid (t, v) \in e'(c)\} \\
& \quad \text{left} = \left\{(t, v) \mid (t, v) \in eS(c) \land \forall (t', -) \in eS'(c). t \neq t' \right\} \\
& \quad \text{both} = \left\{(t, f(v, v')) \mid (t, v) \in eS(c) \land (t', v') \in eS'(c) \land t = t' \right\} \\
& \quad \text{right} = \left\{(t, v) \mid (t, v) \in eS'(c) \land \forall (t', -) \in eS(c). t \neq t' \right\} \\
\end{align*}
\]

\[\text{in left} \cup \text{both} \cup \text{right}\]

\[
\begin{align*}
[\text{union}] \left( \lambda c. \{(\text{delay}_{C \rightarrow S}(t, c), v) \mid (t, v) \in e(c)\}, \right.
& \quad \left. \lambda c. \{(\text{delay}_{C \rightarrow S}(t, c), v) \mid (t, v) \in e'(c)\}, \right. \\
& \quad f \left. \right) = [\text{union}]^{-1} \\
[\text{union}]([\text{toServer}]\langle e \rangle, [\text{toServer}]\langle e' \rangle, f) & = [\text{toServer}] \\
\end{align*}
\]
Bibliography


Publications

International Conference and Workshop Papers


