A Survey of Multitier Programming

PASCAL WEISENBURGER, Technische Universität Darmstadt, Germany
JOHANNES WIRTH, Technische Universität Darmstadt, Germany
GUIDO SALVANESCHI, Technische Universität Darmstadt, Germany

Multitier programming deals with developing the components that pertain to different *tiers* in the system (e.g., client and server), mixing them in the same compilation unit. In this paradigm, the code for different tiers is then either generated at run time or it results from the compiler splitting the codebase into components that belong to different tiers based on user annotations, static analysis, types, or a combination of these. In the Web context, multitier languages aim at reducing the distinction between client and server code, by translating the code that is to be executed on the clients to JavaScript or by executing JavaScript on the server, too. Ultimately, the goal of the multitier approach is to improve program comprehension, simplify maintenance and enable formal reasoning about the properties of the *whole* distributed application.

A number of multitier research languages have been proposed over the last decade, which support various degrees of multitier programming and explore different design trade-offs. In this paper, we provide an overview of the existing solutions, discuss their positioning in the design space and outline open research problems.

CCS Concepts: • **General and reference** → **Surveys and overviews**; • **Software and its engineering** → **Distributed programming languages**; Domain specific languages; • **Theory of computation** → *Distributed computing models.*

Additional Key Words and Phrases: Multitier Languages, Tierless Languages, Distributed Programming

ACM Reference Format:
Pascal Weisenburger, Johannes Wirth, and Guido Salvaneschi. 2020. A Survey of Multitier Programming. *ACM Comput. Surv.* 1, 1, Article 1 (January 2020), 34 pages. https://doi.org/10.1145/3397495

1 INTRODUCTION

Developing distributed systems is widely recognized as a complex and error-prone task. A number of aspects complicate programming distributed software, including concurrent execution on different nodes, the need to adopt multiple languages or runtime environments (e.g., JavaScript for the client and Java for the server), and the need to properly handle complex communication patterns considering synchronicity/asynchronicity, consistency as well as low-level concerns such as data serialization and format conversion. Over the years, developers and practitioners have tackled these challenges with methods that operate at different levels. Various middlewares abstract over message propagation (e.g., Linda [48]). Primitives for remote communication (RPC, e.g., CORBA [50], RMI [107]) give programmers the illusion of distribution transparency. Decoupling in the software architecture improves concurrency and fault tolerance (e.g., the Actor model [56]). Finally,

Authors' addresses: Pascal Weisenburger, Technische Universität Darmstadt, Hochschulstraße 10, Darmstadt, Hessen, 64289, Germany, weisenburger@cs.tu-darmstadt.de; Johannes Wirth, Technische Universität Darmstadt, Hochschulstraße 10, Darmstadt, Hessen, 64289, Germany, johannes-wirth@posteo.de; Guido Salvaneschi, Technische Universität Darmstadt, Hochschulstraße 10, Darmstadt, Hessen, 64289, Germany, salvaneschi@cs.tu-darmstadt.de.

0360-0300/2020/1-ART1 $15.00
https://doi.org/10.1145/3397495

out-of-the-box specialized frameworks can manage fault recovery, scheduling and distribution automatically (e.g., MapReduce [38]).

A radically innovative solution has been put forward by the so-called *multitier programming* (MT) approach (sometimes referred to as *tierless programming*). MT programming consists of developing the components that pertain to different *tiers* in the system (e.g., client and server), mixing them in the same compilation unit. Code for different tiers is generated at run time or split by the compiler into components that belong to different tiers based on user annotations and static analysis, types or a combination of these.

A number of MT *research* languages have been proposed over the last decade, demonstrating the advantages of this paradigm (e.g., [12, 28, 32, 110]), including improving software comprehension, enhancing software design, enabling formal reasoning and ameliorating maintenance. In parallel, a number of *industrial* solutions include concepts from MT programming (e.g., [9, 13, 119]), showing that this approach has great potential in practice.

The success of the MT paradigm has led to a variety of solutions that occupy different points in the design space. These solutions mix techniques (e.g., compile time vs. run time splitting) and design choices (e.g., placement of compilation units vs. placement of single functions) that often depend on the application domain as well as on the software application stack. As a result, it is hard to get a complete picture of the existing trade-offs based on a precise taxonomy of the available design decisions. In this paper, we fill this gap, providing researchers and practitioners with an overview of MT languages and of the fundamental design decisions that this paradigm entails. After presenting a selection of influential MT languages, we systematically analyze existing MT approaches along various axes, highlighting the most important achievements for each language. Finally, we provide an overview of related research areas and of the open research challenges in the field.

This paper is structured as follows. Section 2 introduces MT programming. Section 3 presents concrete examples of MT programming languages to implement a reference application. Section 4 discusses existing MT languages according to our analysis axes. Section 5 provides an overview of open research issues in the area. Section 6 presents approaches that are closely related to MT programming. Section 7 concludes.

2 MULTITIER PROGRAMMING IN A NUTSHELL

The different components of a distributed application are executed on different tiers, where each tier can run on a different machine in a network. For example, a 3-tier (or 3-layer) application is organized into three major parts – usually *presentation*, *application processing*, and *data management* – residing in different network locations [19]. One of the advantages of this approach is that, by organizing a system into tiers, the functionality that is encapsulated into one of the tiers can be modified independently, instead of redesigning the entire application.

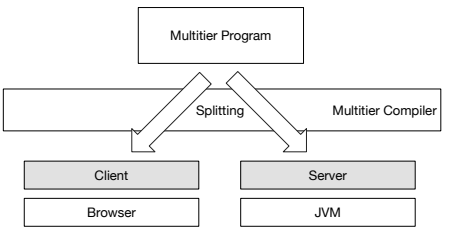

Fig. 1. Multitier Programming.

As a result of this architectural choice, however, a crosscutting functionality that belongs to multiple tiers is separated among several compilation units. For example, in the Web setting, functionality is often scattered across client and server. Also, in many cases, each layer is implemented in a different programming language depending on the technology of the underlying layer, e.g., JavaScript for the browser-based interface, Java for the server-side application logic and SQL for the database.

In an MT programming language, a single language can be used to program different tiers, often adopting different compilation backends based on the target tier (e.g., JavaScript for the browser, Java for the server). As a result, a functionality that spans over multiple tiers can be developed within the same compilation unit. The compiler takes care of generating multiple deployable units (Figure 1) starting from a single MT program as well as of generating the communication code that is required for such modules to interact during program execution.

2.1 Benefits of Multitier Programming

In this section, we provide an overview of the main advantages offered by the MT language design. We report the main claims found in literature and refer to the sources where these are discussed.

2.1.1 Higher Abstraction Level. An important advantage of MT programming is that it enables abstracting over a number of low-level details relevant to programming distributed systems. As a result, software development is simplified and programmers can work at a higher level of abstraction [132]. There are different aspects to consider. First, developers do not face the issue of dealing with error-prone aspects like network communication, serialization, and data format conversions between different tiers [101]. Second, with MT programming, there is no need to design the inter-tier APIs, for example specifying the REST API a server exposes to clients. The technologies used for inter-tier communication are usually transparent to the developer [110] and a detail of the compilation approach.

2.1.2 Improved Software Design. In many distributed applications, the boundaries between hosts and the boundaries between functionalities do not necessarily coincide, i.e., a single functionality can span multiple locations and a single location can host multiple functionalities. For example, retrieving a list of recent emails requires a search on the server, filtering the result on the client and displaying the result. All these operations conceptually pertain to the same functionality. Programming each location separately may result in two design issues. First, it can compromise modularity because functionality (e.g., email retrieval) is scattered across the codebases of different hosts. Second, it is error-prone because of code repetition. For example, encryption requires encrypting and decrypting data on both ends of the communication channel, and the associated functions need to be available on both the client and the server. In contrast, MT programming allows for developing a functionality once and then place it where required [40].

2.1.3 Formal Reasoning. Formal reasoning can benefit from MT design because MT languages model distributed applications as a whole as well as reify a number of aspects of distributed software that are usually left implicit, like placement, components of the distributed system, and the boundaries among tiers. Hence, it becomes easier to formally reason about software properties considering the whole system at once instead of each component in isolation. For example, researchers have developed methods to reason about concurrency [90] and security [10] considering information flow in the whole system. Also, performance can be improved by eliminating dynamic references of global pointers [25]. Finally, researchers considered domain-specific properties, such as reachability in software defined networks via verification [89].

2.1.4 Code Maintenance. MT programming simplifies the process of modifying an existing software system. Two cases are particularly interesting for MT. First, migrating functionality among different tiers does not require a complete rewrite in a different language [49]. For example, validating user input should already happen on the client-side to improve usability and must happen on the server to enforce input validation before further processing. Both validation functions share the same code. Second, it is easier to migrate an application among different platforms [43]. For example, in

principle, the client-side logic of a client–server desktop application can be migrated to the Web just by changing the compilation target of the client side to JavaScript.

2.1.5 Program Comprehension. Program comprehension refers to the complexity (time, required expertise) that a developer faces to come up with a correct mental model of the behavior of a program [116]. A crucial advantage of MT programming is that it simplifies reasoning about data flow over multiple hosts because data flows that belong to a certain functionality are not interrupted by the modularization across the tier axis and by the details of communication code – simplifying development as well as debugging [81]. We are, however, not aware of empirical studies or controlled experiments that measure the advantage of MT programming in terms of program comprehension.

2.2 An Overview of Multitier Languages

In this survey, we compare MT languages, i.e., languages that support implementing different tiers of a distributed system within a single compilation unit. This survey focuses on *homogeneous* MT programming, where tiers follow the same model of computation and have similar processing capabilities. Databases are an example for a tier with a computational model that is typically different from the one of the tier that accesses the database, such as a web server. For MT languages that support *heterogeneous* tiers, such as databases, we only briefly describe the language features that are supported. Table 1 lists the MT approaches we discuss systematically and related approaches on which we touch to point out their connection to MT programming.

Multitier Languages. In this paper, we first show the implementation of a small application (Section 3) in a representative selection of MT languages. These include two languages that pioneered MT programming for the web (Hop/Hop.js and Links), two recent approaches focusing on web development (Ur/Web and Eliom), an approach that also supports more general distributed systems than web applications (ScalaLoci) and Google's GWT, an industrial solution for cross compilation to different tiers, that, however, provides no specific MT abstractions. We then conduct a systematic feature comparison (Section 4) among homogeneous MT languages (first segment of Table 1).

In this survey, we also include programming frameworks that target distributed applications where several tiers are developed together, using the same language (second segment of Table 1). For example, such frameworks reuse existing (non-MT) languages and communication libraries, compiling to JavaScript for the client-side (GWT), using JavaScript for both the client and the server (Meteor) or use an external configuration file for specifying the splitting (J-Orchestra). In these languages, the presence of different tiers is clearly visible to the programmer either in the form of configuration files or source annotations.

Related Approaches. In this survey, we also elaborate on closely related approaches (third segment of Table 1) that do not completely fit the programming model of the aforementioned MT languages and the taxonomy of our feature comparison. Hence, we do not classify them systematically but highlight their connection to MT programming where they relate to the discussed MT aspects. Such approaches (a) do not express tiers as part of their language abstractions because the code is assigned to tiers transparently (Distributed Orc, Jif/split and Fission). In this group, we also include Hiphop, where the language extends an MT language but the extension itself does not add any MT abstraction, and SIF, which uses GWT for JavaScript compilation as well as a client runtime library, and WebDSL, where the language only represents the state of the data model. Other approaches do not completely fit the MT programming model that we consider because they (b) do not include cross-tier communication, intentionally leaving remote communication

Table 1. Overview of MT Languages

Language	Short Description
Hop/Hop.js [110, 111]	Dynamically typed language for developing web applications with a client–server communication scheme and asynchronous callbacks.
Links [32, 46]	Statically typed language that covers the client tier, the server tier and the access to the database tier. It uses remote calls and message passing for client–server communication.
Ur/Web [28]	ML-like language with support for type-safe metaprogramming that provides communication from client to server through remote procedure calls and from the server to the client through message-passing channels.
Eliom/Ocsigen [9, 101]	OCaml dialect that extends the ML module system to support MT modules featuring separate compilation; used in the Ocsigen project.
ScalaLoci [132]	Supports generic distributed systems, not only web applications, thanks to placement types; features remote procedures and reactive programming abstractions for remote communication.
StiP.js [95, 96]	Allows developers to annotate the code that belongs to the client or to the server; slicing detects the dependencies between the annotated fragment and the rest of the code.
Gavial [104, 105]	Domain-specific language embedded into Scala that provides reactive programming abstractions for cross-tier communication.
Opa [102]	Statically typed language that supports remote communication via remote procedure calls and message-passing channels.
AmbientTalk/R [24, 39]	Targets mobile applications with loosely coupled devices and provides reactive programming abstractions on top of a publish–subscribe middleware.
ML5 [88]	Represents different tiers by different possible worlds, as known from modal logic.
WebSharper [13]	Allows developers to specify client-side members and members that are callable remotely.
Haste [41]	Uses monadic computations wrapping client and server code into different monads and provides explicit remote calls.
Fun [135]	Enables automatic synchronization of data across web clients without manually implementing the communication with the server.
Koka [73]	Supports splitting code among tiers using a type and effect system by associating different effects to different tiers.
Multi-Tier Calculus [90]	Provides a formal model to reason about the splitting of MT code into a client and a server part and the communication between both parts through message channels.
Swift [30]	Splits an application into client and server programs based on the flow of private data, making sure that private data does not flow to untrusted clients.
Volta [81]	Uses attributes to annotate classes with the tier they belong to, automatically converting cross-tier method calls to remote invocations.
GWT [64]	Compiles Java to JavaScript for the client and provides remote procedures for client–server communication; developed at Google.
Meteor [119]	A programming framework to use JavaScript for both the client and the server code; provides remote procedures, publish–subscribe abstractions and shared state.
J-Orchestra [122]	Uses configuration files to assign Java classes to tiers, rewriting the Java bytecode to turn method invocations into remote calls.
Hiphop [12]	Extends Hop with synchronous data flows, focusing on guarantees on time and memory bounds.
Distributed Orc [121]	The runtime optimizes the placements of values; it provides location transparency by giving local and remote operations the same semantics, which allows for handling asynchrony and failures uniformly.
Jif/split [141]	Splits a program into tiers based on the flow of private data, making sure that private data do not flow to another tier.
Fission [52]	Dynamically splits a program execution into client-side and server-side execution based on the flow of private data, making sure that private data does not flow to untrusted clients.
SIF [31]	Checks the flow of private data in a web application, making sure that private data does not flow to untrusted clients.
WebDSL [49]	Domain-specific language for specifying the data model of web applications and the web pages to view and edit data model objects.
Acute [113]	Supports type-safe marshalling for remote interaction, versioning of program code and dynamic code reloading, leaving the network communication mechanism to libraries.
Mobl [55]	Supports different concerns of developing the client-side of web applications, such as the data model, the application logic and the user interface.
High-Level Abstractions for Web Programming [106]	Provides a Scala EDSL that captures common tasks performed in web applications, e.g., defining DOM fragments.

Listing 1. Echo application in Hop.js.

```
1  service echo() {
2    var input = <input type="text" />
3    return <html>
4      <body onload=~{
5        var ws = new WebSocket("ws://localhost:" + ${hop.port} + "/hop/ws")
6        ws.onmessage = function(event) { document.getElemenetById("list").appendChild(<li>${event.data}</li>) }
7      }>
8        <div>
9          ${input}
10         <button onclick=~{ ws.send(${input}.value) }>Echo!</button>
11       </div>
12       <ul id="list" />
13     </body>
14   </html>
15 }
16
17 var wss = new WebSocketServer("ws")
18 wss.onconnection = function(event) {
19   var ws = event.value
20   ws.onmessage = function(event) { ws.send(event.value) }
21 }
```

support to libraries, such as Acute and several languages for web applications Mobl, High-Level Abstractions for Web Programming.

MT development shares with cross-compilation the goal of abstracting over different tiers as cross compilation abstracts over the heterogeneity of different target platforms. Cross-compilers include, e.g., Haxe or the Kotlin language, the JSweet Java to JavaScript compiler, the Bridge.NET and the SharpKit C# to JavaScript compilers, and the Scala.js Scala to JavaScript compiler. Yet, these solutions do not offer specific language-level support for distribution and remote communication. This survey discusses the difference between cross-compilers and MT languages, but it does not consider cross-compilers in detail.

3 A GLIMPSE OF MULTITIER LANGUAGES

In this section, we present languages that have pioneered MT programming and/or have been very influential in recent years. To provide an intuition of how MT programming looks like using those languages, we present the same example implemented in each language. As an example, we show an *Echo* client–server application: The client sends a message to the server and the server returns the same message to the client, where it is appended to a list of received messages. The application is simple and self-contained, and – despite all the limitations of short and synthetic examples – it gives us the chance to demonstrate different MT languages side by side.

3.1 Hop

Hop [110] is a dynamically typed Scheme-based language. It follows the traditional approach of modeling communication between client and server using asynchronous callbacks for received messages and return values. JavaScript code is generated at run time and passed to the client. A recent line of work has ported the results of Hop to a JavaScript-based framework, Hop.js [111], which allows using JavaScript to program both the client and the server side.

Listing 1 shows the Echo application implemented in Hop.js. HTML can be embedded directly in Hop code. HTML generated on the server (Line 2–14) is passed to the client. HTML generated on the client can be added to the page using the standard DOM API (Line 6).

Listing 2. Echo application in Links.

```
1  fun echo(item) server {
2    item
3  }
4
5  fun main() server {
6    page
7      <html>
8        <body>
9          <form l:onsubmit="{appendChildren(<li>{stringToXml(echo(item))}</li>, getNodeById("list"))}">
10           <input l:name="item" />
11           <button type="submit">Echo!</button>
12         </form>
13         <ul id="list" />
14       </body>
15     </html>
16 }
17
18 main()
```

Hop supports bidirectional communication between a running server and a running client instance through its standard library. In the Echo application, the client connects to the WebSocket server through the standard HTML5 API (Line 5) and sends the current input value (Line 10). The server opens a WebSocket server (Line 17) that returns the value back to the client (Line 20).

The language allows the definition of *services*, which are executed on the server and produce a value that is returned to the client that invoked the service. For example, the `echo` service (Line 1) produces the HTML page served to the web client of the Echo application. Thus, the code in a service block is executed on the server.

Because of the `~{...}` notation, the code for the `onload` (Line 4) and `onclick` (Line 10) handlers is not immediately executed but the server generates the code for later execution on the client. On the other hand, the `${...}` notation escapes one level of program generation. The expressions `hop.port` (Line 5), `event.data` (Line 6) and `input` (Line 9 and 10) are evaluated by the outer server program and the values to which they evaluate are injected into the generated client program. Hop supports full stage programming, i.e., `~{...}` expressions can be arbitrarily nested such that not only server-side programs can generate client-side programs but also client-side programs are able to generate other client-side programs.

3.2 Links

Links [32] is a statically typed language that translates to SQL for the database tier and to JavaScript for the web browser. The latter is a technique, which was pioneered by the typed query system Kleisli [138] and adopted by Microsoft LINQ [124]. It allows embedding statically typed database queries in Links. Recent work extended Links with algebraic effects [57], provenance tracking [42] and session types [78] with support for exception handling [46]. Links' *Model-View-Update* architecture [45] integrates session typing and GUI development.

Listing 2 shows the Echo application implemented in Links. Links uses annotations on functions to specify whether they run on the client or on the server (Line 1 and 5). Upon request from the client, the server executes the main function (Line 18), which constructs the code that is sent to the client. Links allows embedding XML code (Line 7–15). The `l:name` attribute (Line 10) declares an identifier to which the value of the input field is bound and which can be used elsewhere (Line 9). The code to be executed for the `l:onsubmit` handler (Line 9) is not immediately executed but

Listing 3. Echo application in Ur/Web.

```
1  fun echo (item : string) = return item
2
3  fun main () =
4    let fun mkhtml list =
5      case list of
6        []        => <xml/>
7      | r :: list => <xml><li>{[r]}</li>{mkhtml list}</xml>
8    in
9      item <- source "";
10     list <- source [];
11     return <xml><body>
12       <div>
13         <ctextbox source={item} />
14         <button value="Echo!" onclick={ fn _ =>
15           list' <- get list;
16           item' <- get item;
17           item' <- rpc (echo item');
18           set list (item' :: list')
19         }/>
20       </div>
21       <ul>
22         <dyn signal={
23           list' <- signal list;
24           return (mkhtml list')
25         }/>
26       </ul>
27     </body></xml>
28   end
```

compiled to JavaScript for client-side execution. Curly braces indicate Links code embedded into XML. The `l:onsubmit` handler sends the current input value `item` to the server by calling `echo`. The item is returned by the server and appended to the list of received items using standard DOM APIs. The call to the server (Line 9) does not block the client. Instead, the continuation on the client is invoked when the result of the call is available. Client–server interaction is based on *resumption passing style*: Using continuation passing style transformation and defunctionalization, remote calls are implemented by passing the name of a function for the continuation and the data needed to continue the computation. Rather than of constructing HTML forms manually, like in the example, Links further supports *formlets* [33], an abstraction for composing HTML forms.

To access the database tier, Links features database expressions to represent database connections. For example, to store the list of received items in a server-side database, the expression `table "items" with (item: String) from database "list"` refers to the *items* table in the *list* database that contains records with a single *item* string field. Links supports language constructs for querying and updating databases – such as iterating over records using `for`, filtering using `where` clauses, sorting using `orderby` or applying functions on lists, such as `take` and `drop`, to data sets – which are compiled into equivalent SQL statements.

3.3 Ur/Web

Ur/Web [28] is a language in the style of ML, featuring an expressive type system to support type-safe metaprogramming. The type system ensures correctness of a broad range of properties including (i) validity of generated HTML code, (ii) the types of values of HTML form fields matching the types expected by their handlers or the types of columns of a database table, (iii) validity of SQL queries, (iv) lack of dead intra-application links and (v) prevention of code injection attacks. Remote

Listing 4. Echo application in Eliom.

```
1  module Echo_app = Eliom_registration.App (struct let application_name = "echo" let global_data_path = None end)
2
3  let%server main_service = create ~path:(Path []) ~meth:(Get Eliom_parameter.unit) ()
4
5  let%server make_input up =
6    let inp = Html.D.Raw.input () in
7    let btn = Html.D.button ~a:[Html.D.a_class ["button"]] [Html.D.pcdata "Echo!"] in
8    ignore [%client
9      (Lwt.async (fun () ->
10       Lwt_js_events.clicks (Html.To_dom.of_element ~%btn) (fun _ _ ->
11         ~%up (Js.to_string (Html.To_dom.of_input ~%inp)##.value);
12         Lwt.return_unit)) : unit) ];
13   Html.D.div [inp; btn]
14
15 let%server () = Echo_app.register
16   ~service:main_service
17   (fun () () ->
18     let item_up = Up.create (Eliom_parameter.ocaml "item" [%derive.json :string]) in
19     let item_down = Down.of_react (Up.to_react item_up) in
20     let list, handle = ReactiveData.RList.create [] in
21     let list = ReactiveData.RList.map [%shared fun i -> Html.D.li [Html.D.pcdata i] ] list in
22     let input = make_input item_up in
23     ignore [%client
24       (Eliom_client.onload
25         (fun _ -> ignore (React.E.map (fun i -> ReactiveData.RList.cons i ~%handle) ~%item_down)) : unit) ];
26       Lwt.return (Eliom_tools.D.html ~title:"echo" (Html.D.body [input; Html.R.ul list])))
```

procedure calls are executed atomically, with Ur/Web guaranteeing the absence of observable interleaving operations.

Listing 3 shows the Echo application implemented in Ur/Web. Ur/Web allows embedding XML code using `<xml>...</xml>` (Line 6 and 7). The {...} notation embeds Ur/Web code into XML. {[...]} evaluates an expression and embeds its value as a literal. Ur/Web supports functional reactive programming for client-side user interfaces. The example defines an `item` source (Line 9), whose value is automatically updated to the value of the input field (Line 13) when it is changed through user input, i.e., it is *reactive*. The `list` source (Line 10) holds the list of received items from the echo server. Sources, time-changing input values, and signals, time-changing derived values, are Ur/Web's reactive abstractions, i.e., signals recompute their values automatically when the signals or sources from which they are derived change their value, facilitating automatic change propagation. Upon clicking the button, the current value of `list` (Line 15) and `item` is accessed (Line 16), then a remote procedure call to the server's `echo` function is invoked (Line 17) and `list` is updated with the item returned from the server (Line 18). To automatically reflect changes in the user interface, a signal is bound to the `signal` attribute of the HTML pseudo element `<dyn>` (Line 22). The signal uses the `mkhtml` function (Line 24, defined in Line 4), which creates HTML list elements. In addition to remote procedure calls – which initiate the communication from client to server – Ur/Web supports typed message-passing channels, which the server can use to push messages to the client.

Ur/Web integrates a domain-specific embedding of SQL for accessing the database tier with clauses such as `SELECT`, `FROM` or `ORDERBY`. For example, a set of database records storing the list of received items is specified by a `table items : { item : string }` declaration. Such `table` declarations can be private to a module using an ML-style module system for encapsulating database tables.

Listing 5. Echo application in GWT.

```
a.1 package echo.client;
a.2 public interface EchoService extends RemoteService {
a.3   String echo(String item) throws IllegalArgumentException; }

b.1 package echo.client;
b.2 public interface EchoServiceAsync {
b.3   void echo(String item, AsyncCallback<String> callback) throws IllegalArgumentException; }

c.1 package echo.server;
c.2 public class EchoServiceImpl extends RemoteServiceServlet implements EchoService {
c.3   public String echo(String item) throws IllegalArgumentException {
c.4     return item; } }

d.1 package echo.client;
d.2 public class Echo implements EntryPoint {
d.3   private final EchoServiceAsync echoService = GWT.create(EchoService.class);
d.4
d.5   public void onModuleLoad() {
d.6     final TextBox itemField = new TextBox();
d.7     final Button submitButton = new Button("Echo!");
d.8
d.9     RootPanel.get("itemFieldContainer").add(itemField);
d.10    RootPanel.get("submitButtonContainer").add(submitButton);
d.11
d.12    submitButton.addClickHandler(new ClickHandler {
d.13      public void onClick(ClickEvent event) {
d.14        echoService.echo(itemField.getText(), new AsyncCallback<String>() {
d.15          public void onFailure(Throwable caught) { }
d.16          public void onSuccess(String result) {
d.17            RootPanel.get("itemContainer").add(new Label(result)); } }); } }); } }
```

3.4 Eliom

Eliom [101] is an OCaml dialect designed in the context of the Ocsigen project [9] for developing client–server web applications. Ocsigen further provides mechanisms to support a number of practical features necessary in modern applications, including session management and bidirectional client–server communication through its standard library.

Listing 4 shows the Echo application in Eliom. Eliom extends `let`-bindings with *section annotations* `%client`, `%server` and `%shared` – the latter indicates code that runs on both the client and the server. The application starts with a call to `Echo_app.register` (Line 15). Eliom supports cross-tier reactive values: The application generates a server-side event (Line 18) and a corresponding client-side event (Line 19), which automatically propagates changes from the server to the client. A reactive list (Line 20) holds the items received from the server. Mapping the list produces a list of corresponding HTML elements (Line 21), which can directly be inserted into the generated HTML code (Line 26). Eliom supports a DSL for HTML, providing functions of the same name as the HTML element they generate. Server-side code can contain nested *fragments* to be run on the client (`[%client …]`, Line 23) or to be run on both the client and the server (`[%shared …]`, Line 21). Eliom uses *injections* (prefixed by `~%`) to access values on the client side that were computed on the server. The client-side representation of the event `item_down` is injected into a client fragment to extend the reactive list with every item returned from the server (Line 25). The `make_input` function (Line 5) generates the main user interface, which processes the stream of button clicks (Line 10) and fires the `up` event for every item (Line 11). To fire the server-side `up` event from the client-side, we inject the event via `~%up` into the client fragment.

Listing 6. Echo application in ScalaLoci.

```
1  @multitier object Application {
2    @peer type Server <: { type Tie <: Single[Client] }
3    @peer type Client <: { type Tie <: Single[Server] }
4
5    val message = on[Client] { Event[String]() }
6    val echoMessage = on[Server] { message.asLocal }
7
8    def main() = on[Client] {
9      val items = echoMessage.asLocal.list
10     val list = Signal { ol(items() map { message => li(message) }) }
11     val inp = input.render
12     dom.document.body = body(
13       div(
14         inp,
15         button(onclick := { () => message.fire(inp.value) })("Echo!")),
16       list.asFrag).render
17   }
18 }
```

3.5 Google Web Toolkit (GWT)

GWT [64] is an open source project developed at Google. Its design has been driven by a pragmatic approach, mapping traditional Java programs to web applications. A GWT program is a Java Swing application except that the source code is compiled to JavaScript for the client side and to Java bytecode for the server side. Compared to fully-fledged MT programming, distributed code in GWT is not developed in a single compilation unit nor necessarily in the same language. Besides Java, in practice, GUIs often refer to static components in external HTML or XML files. Client and server code reside in different Java packages. GWT provides RPC library support for cross-tier communication.

Listing 5 shows the Echo application implemented in GWT. For the sake of brevity, we leave out the external HTML file. The application adds an input field (Line d.9) and a button (Line d.10) to container elements defined in the HTML file and registers a handler for click events on the button (Line d.12). When the button is clicked, the `echo` method of the `echoService` is invoked with the current item and a callback – to be executed when the remote call returns. When an item is returned by the remote call, it is added to the list of received items (Line d.17). GWT requires developers to specify both the interface implemented by the service (Line a.2) and the service interface for invoking methods remotely using a callback (Line b.2). The implementation of the echo service (Line c.2) simply returns the item sent from the client.

3.6 ScalaLoci

ScalaLoci [132] is a language that targets *generic* distributed systems rather than the Web only, i.e., it is not restricted to a client–server architecture. To this end, ScalaLoci supports peer types to encode the different locations at the type level. Placement types are used to assign locations to data and computations. ScalaLoci supports multitier reactives – language abstractions for reactive programming that are placed on specific locations – for composing data flows cross different peers.

Listing 6 shows the Echo application implemented in ScalaLoci. The application first defines an input field (Line 11) using the ScalaTags library [76]. The value of this input field is used in the click event handler of a button (Line 15) to fire the `message` event with the current value of the input field. The value is then propagated to the server (Line 6) and back to the client (Line 9). On the client, the values of the event are accumulated using the `list` function and mapped to an HTML list (Line 10). This list is then used in the HTML code (Line 16) to display the previous inputs.

4 ANALYSIS

In this section we systematically analyze existing MT solutions along various axes. We consider the following dimensions:

- *Degrees of MT programming* refers to the amount of MT abstractions supported by the language. At one extreme of the spectrum, we find languages with dedicated MT abstractions for data sharing among tiers and for communication. At the other end of the spectrum lie languages where part of the codebase can simply be cross-compiled to a different target platform (e.g., Java to JavaScript) to enhance the interoperability between tiers but do not provide specific MT abstractions.
- *Placement strategy* describes how data and computations in the program are assigned to the hosts in the distributed system, e.g., based on programmers' decisions or based on automatic optimization.
- *Placement specification and granularity* in MT languages refers to the means offered for programmers to specify placement (e.g., code annotations, configuration files) and their granularity level (e.g., per function, per class).
- *Communication abstractions* for communication among tiers are a crucial aspect in MT programming since MT programming brings the code that belongs to different tiers to the same compilation unit. MT approaches provide dedicated abstractions to simplify implementing remote communication which differ considerably among languages.
- *Formalization of MT languages* considers the approach used to formally define the semantics of the language and formally prove properties about programs.
- *Distribution topologies* describe the variety of distributed architectures [47] (e.g., client–server, peer-to-peer) that a language supports.

4.1 Degrees of MT Programming

Several programming frameworks for distributed systems have been influenced, to various degrees, by ideas from MT. In this section, we compare languages where MT programming is supported by dedicated abstractions, either by explicitly referring to placement in the language or by using scoping in the same compilation unit to define remote communication, and approaches that share similar goals to MT programming using compilation techniques that support different targets (and tiers), but do not expose distribution as a language feature to the developer. Table 2 provides an overview of existing solutions concerning the degree of supported MT programming. Specifically, it considers support for cross compilation and the supported language features for distribution.

Multitier distribution provides a programming model that defines different tiers and offers abstractions for developers to control the distribution.

Transparent distribution does not support code assignment to tiers as a reified language construct. Splitting into tiers is computed transparently by the compiler or the runtime and not part of the programming model.

No distribution abstractions do not provide language features specific to the distribution of programs.

When running distributed applications on different machines, approaches related to MT programming either assume the same execution environment, where all tiers can be supported by a **uniform compilation** scheme, or employ a **cross compilation** approach to support different target platforms. Cross-compilers can be used to support the development of distributed systems (e.g., by compiling client-side code to JavaScript) but still require manual distribution of code and do not offer abstractions for remote communication among components as MT languages do. Traditional languages, falling into the bottom right corner of Table 2, neither support distribution

Table 2. Degrees of MT Programming

Compilation Approach	**Distribution Approach**		
	Distribution		*No Distribution Abstractions*
	Multitier	*Transparent*	
Cross Compilation	Hop		Haxe
	Links		Kotlin
	Opa		JSweet
	Ur/Web		Bridge.NET
	Eliom/Ocsigen		SharpKit
	Gavial		Scala.js
	ML5		WebDSL
	ScalaLoci		Mobl
	WebSharper		High-Level Abstractions
	Haste		for Web Programming
	Swift		
	Volta		
	GWT		
Uniform Compilation	Hop.js	Distributed Orc	Hiphop
	StiP.js	Jif/split	SIF
	AmbientTalk/R	Fission	Acute
	Fun		(traditional languages)
	Koka		
	Multi-Tier Calculus		
	J-Orchestra		
	Meteor		

nor cross compilation. Hiphop [12] does not provide its own support for distribution but relies on Hop's [110] MT primitives. SIF [31] uses information flow control to ensure that private data does not flow to untrusted clients. It is implemented on top of Java Servlets, which respond to requests sent by web clients. Acute [113] is an OCaml extension that, although it does not support distribution or cross compilation, provides type-safe marshalling for accessing resources remotely based on transmitting type information at run time for developing distributed systems.

We provide examples for the multitier category, which is extensively discussed in the rest of the paper, and systematically analyze the second and third approach using transparent splitting by the compiler or manual splitting and cross compilation, respectively.

4.1.1 Dedicated MT Programming Abstractions. MT languages provide abstractions that reify the placement of data and computations and allow programmers to directly refer to these concepts in their programs. In Hop.js [111], inside the same expression, it is possible to switch between server and client code with `~{...}` and `${...}`, which can be arbitrarily nested. Similarly, the Ur/Web [28] language provides the `{...}` escape operator. In ScalaLoci [132], placement is part of the type system (placement types) and the type checker can reason about resource location in the application. Eliom's [101] placement annotations `%client`, `%server` and `%shared` allow developers to allocate resources in the program at the granularity of variable declarations. Similarly, Links [32] provides a `client` and a `server` annotation to indicate functions that should be executed on the client or the server, respectively.

The MT languages above hide the mismatch between the different platforms underlying each tier, abstracting over data representation, serialization and network protocols, enabling the combination of code that belongs to different tiers within the same compilation unit. In addition, MT concepts

are reified in the language in the sense that language abstractions enable developers to refer to tiers *explicitly*.

4.1.2 Compilers for Multitier Programming. Transparent distribution approaches enable using a single language for different tiers and support compilation to tier-specific code, but do not provide specific abstractions for MT programming. Splitting a program into different tiers based on security concerns (Jif/split [141], Fission [52]) adopts information flow control techniques to ensure that private data does not leak to untrusted tiers. Distributed Orc [121] automatically optimizes the distribution of values at runtime to minimize communication cost.

Approaches that add compilation to a different platform for existing general-purpose languages have been proposed by different vendors and organizations, targeting various languages and programming platforms, e.g., the JSweet Java to JavaScript compiler, the Bridge.NET and the SharpKit C# to JavaScript compilers and the Scala.js Scala to JavaScript compiler. Haxe [43] is a cross-platform toolkit based on the statically typed object-oriented Haxe language that compiles to JavaScript, PHP, C++, Java, C#, Python and Lua. The statically typed language Kotlin [62] for multi-platform applications targets the JVM, Android, JavaScript and native code. Such approaches do not support *automatic* separation into tiers – the developer has to keep the code for different tiers separate, e.g., in different folders. Remote communication APIs are provided by libraries depending on the target platform (e.g., TCP sockets or HTTP). Such solutions are the most pragmatic: They do not break compatibility with tooling – if already available – and provide a programming model that is quite close to traditional programming. Developers do not significantly change the way they reason about coding distributed applications and do not need to learn completely new abstractions.

Domain-specific languages take over tasks specific to (certain types of) distributed applications, such as constructing a client-side user interface based on a given data model. Richard-Foy et al. [106] propose a Scala EDSL that captures common tasks performed in web applications, e.g., defining DOM fragments. Their approach allows specializing code generation depending on the target platform, e.g., using the Scala XML library when compiling to Java bytecode or using the browser's DOM API when compiling to JavaScript. Mobl [55] is a DSL for building mobile web applications in a declarative way providing language features for specifying the data model, the application logic and the user interface. Mobl compiles to a combination of different target languages, HTML, CSS and JavaScript. It, however, targets the client side only.

4.2 Placement Strategy

The placement strategy is the approach adopted by MT languages to assign data and computations in the program to the hosts comprising the distributed system. Table 3 classifies MT languages into approaches where placement is done **automatically** and approaches where placement is **explicitly** specified by the developer. Even for MT solutions with automatic placement, the assignment to different hosts is an integral part of the programming model. For example, specific parts of the code have a fixed placement (e.g., interaction with the

Table 3. Placement Strategy

Language	Placement Strategy	
	Automatic	Explicit
Hop/Hop.js	·	staged
Links	·	partitioned
Opa	partitioned	·
StiP.js	partitioned	·
Ur/Web	·	staged
Eliom/Ocsigen	·	staged
Gavial	·	partitioned
AmbientTalk/R	·	partitioned
ML5	·	partitioned
ScalaLoci	·	partitioned
WebSharper	·	partitioned
Haste	·	partitioned
Fun	·	partitioned
Koka	·	partitioned
Multi-Tier Calculus	partitioned	·
Swift	partitioned	·
Volta	·	partitioned
J-Orchestra	·	partitioned
Meteor	·	partitioned
GWT	·	partitioned

web browser's DOM must be on the client) or the developer is given the ability to use location annotations to enforce a certain placement.

The code that is assigned to different places is either (1) **partitioned** (statically or dynamically) into different programs or (2) separated into different **stages**, where the execution of one stage generates the next stage and can inject values computed in the current stage into the next one. When accessing a value of another partition in approach (1), the value is looked up remotely over the network and the local program continues execution with the remote value after receiving it. For handling remote communication asynchronously, remote accesses are either compiled to continuation-passing style or asynchronicity is exposed to the developer using local proxy objects such as futures. Using approach (2) for web applications, the server stage runs and creates the program to be sent to the client. When generating the client program, references to server-side values are spliced into client code, i.e., the client program that is sent already contains the injected server-side values. Such staged execution reduces communication overhead since server-side values accessed by the client are already part of the generated client program.

In the case of web applications, as response to an HTTP request, the server delivers the program to the client which executes it in the browser. For MT languages that do not target web applications, the programs that result from the splitting start independently on different hosts and connect to other parts upon execution, e.g., using peer-to-peer service discovery in AmbientTalk/R.

We first consider placement based on the different functionalities of the application logic which naturally belong to different tiers. Then we present approaches where there are multiple options for placement and the MT programming framework assigns functionalities to tiers based on various criteria such as performance optimization and privacy.

4.2.1 Placement Based on Functional Properties. In most MT languages, the placement of each functionality is fully defined by the programmer by using an escaping/quoting mechanism (Hop [110], Ur/Web [28], Eliom [101]), annotations (Links [32]) or a type-level encoding (ML5 [88], Gavial [105], ScalaLoci [132]). Placement allows separate parts of the MT program to execute on different hosts. The compile-time separation into different components either relies on (whole-)program analysis (Ur/Web, ML5) or supports modular separation (Eliom, ScalaLoci), where each module can be individually split into multiple tiers. On the other hand, dynamic separation is performed at run time (Links, Hop).

When the placement specification is incomplete there is room for alternative placement choices, in which case slicing [134] detects the dependencies between the fragments manually assigned by developers and the rest of the code base, ultimately determining the splitting border. For example, in StiP.js [95, 96], code fragments are assigned to a tier based on annotations, then slicing uncovers the dependencies. This solution allows developing MT web applications in existing general-purpose languages as well as retaining compatibility with development tools. In the slicing process, placement can be constrained not only explicitly, but also based on values' behavior, e.g., inferring code locations using control flow analysis or rely on elements for which the location is known (e.g., database access takes place on the server, interaction with the DOM takes place on the client) [31, 97, 102]. This complicates the integration into an existing language, especially in presence of effects, and is less precise than explicit annotations – hindering, e.g., the definition of data structures that combine fragments of client code and other data [101].

4.2.2 Placement Strategies. For the functionalities that can execute both on the client and on the server, MT approaches either place unannotated code both on the client and on the server (e.g., Links [32], Opa [102], ScalaLoci [132]) or compute the placement that minimizes the communication cost between tiers (e.g., Distributed Orc [121]). Neubauer and Thiemann [90, 91] allow a propagation strategy to produce different balances for the amount of logic that is kept on the client and on

Table 4. Placement Approach

Language	Placement Specification Approach for given Granularity						
	Expression	Binding	Block	Top-Level Binding	Top-Level Block	Class/Module	File
Hop/Hop.js	escaping/quoting	·	·	annotation	·	·	·
Links	·	·	·	annotation	·	·	·
Opa	·	annotation and static analysis	·	·	·	·	·
StiP.js	·	·	·	·	annotation and static analysis	·	·
Ur/Web	escaping/quoting	·	·	dedicated	·	·	·
Eliom/Ocsigen	escaping/quoting	·	·	annotation	·	annotation	·
Gavial	type	·	·	·	·	·	·
AmbientTalk/R	·	·	·	·	·	dedicated	·
ML5	type	·	·	·	·	·	·
ScalaLoci	type	·	·	type	·	·	·
WebSharper	·	·	·	annotation	·	annotation	·
Haste	type	·	·	·	·	·	·
Fun	·	·	·	dedicated	·	·	·
Koka	·	·	·	type	·	·	·
Multi-Tier Calculus	static analysis	·	·	·	·	·	·
Swift	static analysis	·	·	·	·	·	·
Volta	·	·	·	·	·	annotation	·
J-Orchestra	·	·	·	·	·	external	·
Meteor	·	·	dynamic run time check	·	·	·	directory
GWT	·	·	·	·	·	·	directory

the server, starting the propagation from some predefined operators whose placement is fixed. The propagation strategy uses a static analysis based on location preferences and communication requirements to optimize performance (contrarily to many MT approaches where the choice is left to the programmer). Jif/split [141] considers placement based on security concerns: Protection of data confidentiality is the principle to guide the splitting. The input is a program with security annotations and a set of trust declarations to satisfy. The distributed output program satisfies all security policies. As a result, programmers can write code that is agnostic to distribution, but features strong guarantees on information flow. Similarly, Swift [30] also partitions programs based on security labels, but focuses on the Web domain, where the trust model assumes a trusted server that interacts with untrusted clients.

An exception to the approaches above – which all adopt a *compile time* splitting strategy – is Fission [52], which uses information flow control to separate client and server tiers at run time. The dynamic approach allows supporting JavaScript features that are hard to reason about statically, such as `eval`, as well as retaining better compatibility with tooling.

4.3 Placement Specification and Granularity

Placement specification in MT languages is defined at different granularity levels. Languages that allow composing code belonging to different hosts in the *same compilation unit* follow various approaches to specify the execution location. Table 4 classifies the MT languages based on the placement specification approach (Section 4.3.1) and the granularity given in the first row (Section 4.3.2). For example, Hop.js allows escaping arbitrary expressions to delimit code of a different tier. Links uses annotations on top-level bindings to specify the tier to which a binding belongs.

4.3.1 Placement Specification. We identified the following strategies used by MT languages to determine placement:

Dedicated tier assignment always associates certain language constructs to a tier, e.g., top-level name bindings are always placed on the server or every class represents a different tier.

Annotations specify the tier to which the annotated code block or definition belongs, driving the splitting process.

Escaping/quoting mechanisms are used when the surrounding program is placed on a specific tier, e.g., the server, and nested expressions are escaped/quoted to delimit the parts of the code that run on another specific tier, e.g., the client.

Types of expressions determine the tier, making placement part of the type system.

Static analysis determines the tier assignment at compile-time based on functional properties of the code (such as access to a database or access to the DOM of the webpage).

Dynamic run time checks allow developers to check at run time which tier is currently executing the running code, and select the tier-specific behavior based on such condition.

The following strategies are used by approaches lacking language-level support for placement:

External configuration files assign different parts of the code (such as classes) to different tiers.

Different directories are used to distinguish among the files containing the code for different tiers.

Links [32] and Opa [102] provide dedicated syntax for placement (e.g., `fun f() client` and `fun f() server` in Links). Volta [81] relies on the C# base language's custom attribute annotations to indicate the placement of abstractions (e.g., `[RunAt("Client")] class C`). WebSharper [13] uses a `JavaScript` F# custom attribute to instruct the compiler to translate a .NET assembly, a module, a class or a class member to JavaScript (e.g., `[<JavaScript>] let a = ...`). Stip.js [95] interprets special forms of comments (e.g., `/* @client */ {...}` and `/* @server */ {...}`). While MT languages usually tie the placement specification closely to the code and define it in the same source file, approaches like J-Orchestra [122], require programmers to assign classes to the client and server sites in an XML configuration file.

ML5 [88] captures the placement explicitly in the type of an expression. For example, an expression *expr* of type `string @ server` can be executed from the `home` world using `from server get` *expr*. The placement of every expression is determined by its type and the compiler ensures type-safe composition of remote expressions through `from ... get`. Similarly, in ScalaLoci [132], a binding *value* of type `String on Server` can be accessed remotely using `value.asLocal`. Haste [41] also features a type-based placement specification using monadic computations by wrapping client and server code into different monads. Koka uses a type and effect system to capture which functions can only be executed on the client and which functions can only be executed on the server, preventing cross-tier access without explicitly sending and receiving messages.

4.3.2 Placement Granularity. On a different axis, existing MT approaches cover a wide granularity spectrum regarding the abstractions for which programmers can define placement: **files** (e.g., GWT [64]), **classes** (e.g., Volta [81], J-Orchestra [122]), **top-level code blocks** (e.g., Stip.js [95]), **top-level bindings** (e.g., Links [32]), (potentially nested) **blocks** (e.g., Meteor [119]), **bindings** (e.g., Opa [102]) and **(sub)expressions** (e.g., Eliom [101], ML5 [88]). Specification granularities supported by a language are not mutually exclusive, e.g., ScalaLoci [132] supports placed top-level bindings and nested remote blocks. In Hop [110] and Ur/Web [28], which target web applications, where the execution of server code is triggered by an HTTP client request, all top-level bindings define server-side code and nested client-side code is escaped/quoted at the granularity of expressions. Eliom [101] supports both nested client expressions and annotated top-level client/server bindings.

Table 5. Communication Abstractions

Language	Communication Abstraction				
	Remote Procedures	Message Passing	Publish–Subscribe	Reactive Programming	Shared State
Hop/Hop.js	●$^{c\to s}$	○	○	·	·
Links	●^{t,c}	●	·	·	·
Opa	●^{t}	●	·	·	·
StiP.js	●^{t}	·	●	·	●
Ur/Web	●$^{c\to s}$	●$^{s\to c}$	·	·	·
Eliom/Ocsigen	○	○	·	○	·
Gavial	·	·	·	●	·
AmbientTalk/R	·	·	●	●	·
ML5	●	·	·	·	·
ScalaLoci	●	·	·	●	·
WebSharper	●	·	·	·	·
Haste	●$^{c\to s}$	·	·	·	·
Fun	·	·	·	·	●
Koka	·	●	·	·	·
Multi-Tier Calculus	●^{t}	·	·	·	·
Swift	●^{t}	·	·	·	·
Volta	●^{t}	·	·	·	·
J-Orchestra	●^{t}	·	·	·	·
Meteor	○	·	○	·	○
GWT	○	·	·	·	·

● Language support
○ Support through libraries
$^{c\to s}$ From client to server only
$^{s\to c}$ From server to client only
^{t} Fully transparent remote procedure
^{c} Client-initiated

The approach most akin to traditional languages is to force programmers to define functionalities that belong to different hosts in *separated compilation units* such as different Java packages (GWT [64]) or different directories (Meteor [119]). An even coarser granularity is distribution at the software component level. R-OSGi [103] is an OSGi extension where developers specify the location of remote component loading and Coign [59] extends COM to automatically partition and distribute binary applications. These solutions, however, significantly depart from the language-based approach of MT programming.

4.4 Communication Abstractions

MT approaches provide dedicated abstractions intended to simplify implementing remote communication, which differ considerably among languages. Table 5 provides an overview over these abstractions. Languages either support specific forms of communication only in a single direction – either from client to server or from server to client – or support bidirectional communication (potentially requiring the client to initiate the communication). MT languages also differ in whether they make remote communication explicit (and with it, the associated performance impact) or completely transparent to the developer.

Remote communication mechanisms are either integrated into the language using convenient syntactic constructs (e.g., `from ... get` *`expr`* in ML5 [88], *`value`*`.asLocal` in ScalaLoci [132] or `rpc` *`fun`* in Ur/Web [28]), or are made available through the standard library that comes with the language (e.g., `webSocket.send(`*`message`*`)` in Hop.js [111] or `service.`*`fun`*`(new AsyncCallback() {...})` in GWT [64] or `Meteor.call("`*`fun`*`", function(error, result) {...})` in Meteor [119]). We list the communication approaches found in the respective MT languages in Table 5. Developers can, however, implement such communication mechanisms that are not supported out-of-the-box (by dedicated language features or as part of the standard library) as an external library, e.g., providing a library that supports event-based communication based on remote procedure calls or using a

persistent server (e.g., in Links [32]) to emulate shared data structures. We do not consider such external solutions here. We identify the following remote communication mechanisms:

Remote procedures are the predominant communication mechanism among MT languages. Remote procedures can be called in a way similar to local functions – either completely transparently or using a dedicated remote invocation syntax – providing a layer of abstraction over the network between the call site and the invoked code.

Message passing abstractions are closer to the communication model of the underlying network protocols, where messages are sent from one host to another.

Publish–subscribe allows tiers to subscribe to topics of their interest and receive the messages published by other tiers for those topics.

Reactive programming for remote communication defines data flows across tiers through event streams or time-changing values that upon each change automatically update the derived reactive values on the remote tiers.

Shared state makes any updates to a shared data structure performed on one tier available to other tiers accessing the data structure.

MT languages that target the Web domain follow a traditional request–response scheme, where web pages are generated for each client request and the client interacts with the server by user navigation. Both Hop [110] and Eliom [101] allow client and server expressions to be mixed. All server expressions are evaluated on the server before delivering the web page and client expressions are evaluated on the client. Hop additionally provides traditional client–server communication via asynchronous callbacks, whereas Eliom supports more high-level communication mechanisms based on reactive programming through libraries.

WebDSL [49], for example, is an external DSL for web applications to specify the data model and the pages to view and edit data model objects. HTML code is generated for pages, which is reconstructed upon every client request.

4.4.1 Call-Based communication. MT languages provide communication abstractions for client–server interaction not necessarily related to page loading, including RPC-like calls to remote functions, shared state manipulation or message-passing. Abstracting over calling server-side services and retaining the result via a local callback, Links [32] allows bidirectional remote function calls, between client and server. RPC calls in Links, however, hide remote communication concerns completely which has been criticized because the higher latency is not explicit [63]. In contrast, Links' more recent message-passing communication mechanism features explicit send and receive operations.

In both Ur/Web [28] and Opa [102], server and client can communicate via RPCs or message-passing channels. Due to the asymmetric nature of client–server web applications, Ur/Web follows a more traditional approach based on RPCs for client-to-server communication and provides channels for server-to-client communication.

4.4.2 Event-Based Communication. Publish–subscribe middleware has been used in the context of loosely coupled mobile devices (AmbientTalk [24, 39]). Hiphop [12], which extends Hop [110] with synchronous data flows, borrows ideas from *synchronous data flow languages, à la* Esterel [11]. The approach provides substantial guarantees on time and memory bounds, at the cost, however, of significantly restricting expressivity. In ScalaLoci [132], Gavial [104, 105], AmbientTalk/R [39] or libraries for Eliom [101], tiers expose behaviors (a.k.a. signals) and events in the style of functional reactive programming to each other.

4.4.3 Distributed Shared State. Meteor [119] provides *collections* to store JSON-like documents and automatically propagate changes to the other tier. Similarly, in Fun [135], a language for

real-time web applications, modifications to variables bound to the `Global` object are automatically synchronized across clients. MT languages usually support (or even require) a central server component, enabling shared state via the server as central coordinator that exposes its state to the clients.

4.5 Formalization of MT Languages

From a formal perspective, MT programming has been investigated in various publications. In this section, we first present a classification of existing formal models using three analysis directions: the formalization approach, the proof methods and the properties considered in the formalization. Finally, we describe the formalizations of MT languages in more details, classifying them according to the points above.

4.5.1 Techniques and Scope. Existing formal models for MT languages that specify an operational *semantics* follow three main approaches: (s1) they formalize how a single **coherent MT program** is executed modeling how computation and communication happen in the whole distributed setting (e.g., with a semantics where terms can be reduced at different locations) [16, 90, 101, 132], (s2) they specify a **splitting transformation** that describes how tier-specific programs are extracted from MT code and they provide an independent reduction model for the split tiers [34, 90, 101] or (s3) they specify the semantics in terms of an **existing calculus** [73], i.e., the semantics of a calculus not specific to MT languages is reinterpreted for MT programming, e.g., different effects in a type and effect system represent different tiers. Serrano and Queinnec's [112] continuation-based denotational semantics is an exception to the operational approach. It disregards concurrent execution of client and server focusing on a sequential fragment of Hop to model dynamic server-side client code generation.

Based on the models above, researchers looked at *properties* including (p1) **type soundness** as progress and preservation [16, 73, 90, 132], (p2) **behavioral equivalence** of the execution of the source MT program (cf. a1) and the interacting concurrent execution of the tier-specific programs (cf. a2) [34, 90, 101], and (p3) **domain-specific properties** that are significant in a certain context such as secure compilation [10], or performance for data access [25], as well as domain-specific properties, such as host reachability in software defined networks [89]. Crucially, the fact that MT languages model client and server together enables reasoning about global data flow properties such as privacy. The small-step semantics of Hop [16] has been used to model the browser's same-origin policy and define a type system that enforces it. A similar approach has been proposed to automatically prevent code injection for web applications [80]. Splitting in Swift [30] is guaranteed to keep server-side private information unreachable by client-side programs.

Researchers adopted *proof methods* that belong to two categories: (m1) perform the proofs directly on the semantics that describes the whole system and/or the splitting transformation [16, 34, 90, 101, 132] or (m2) leverage proved properties of an existing calculus [32, 73].

4.5.2 Formalizations. Table 6 provides a classification of the formalizations of MT languages. For the discussion, we leave out languages lacking a formal development. Most formalizations model MT applications as single coherent programs, providing soundness proofs for the MT language. Another common approach for reasoning about the behavior of MT code is to formally define the splitting transformation that separates MT code into its tier-specific parts to show behavioral equivalence of the original MT program and the split programs after the transformation. In the case of Hop [16] formal reasoning focuses on properties specific to the Web domain, e.g., conformance of MT programs to the browser's same-origin policy. Koka's effect system [73] can be used to implement different tiers in the same compilation unit. The sound separation into different tiers in Koka follows from the soundness of the effect system.

Table 6. Formalization Approach

Language	Proved Properties			
	Type Soundness of Coherent MT Program	Type Soundness based on Existing Calculus	Behavioral Equivalence of Splitting Transformation	Domain-Specific Properties
Hop/Hop.js	denotational	·	·	operational (same-origin policy)
Links	·	·	operational	·
Eliom/Ocsigen	operational	·	operational	·
ScalaLoci	operational	·	·	·
Multi-Tier Calculus	operational	·	operational	·
Koka	·	operational	·	·

The seminal work by Neubauer and Thiemann [90] presents an MT calculus for web applications. A static analysis on a simply-typed call-by-value lambda calculus determines which expressions belong to each location and produces the assignment of the code to the locations, which results in a lambda calculus with annotated locations. A further translation to an MT calculus (s1) explicitly models opening and closing of communication channels. Type soundness for the MT calculus is proved (p1). The splitting transformation (s2), which extracts a program slice for each location, is proved to generate only valid programs wrt. the source (p2). The transformed program is considered valid if it is weakly bisimilar [92] to the source program, i.e, if it performs the same operations with the same side effects and the operations are in the same order (m1).

Boudol et al. provide a small-step operational semantics for Hop [16], which covers server-side and client-side computations, concurrent evaluation of requests on the server and DOM manipulation (s1). For Hop, based on Scheme, which does not feature a static type system, the authors define a type system for "request-safety" (p1), which ensures that client code will never request server-side services that do not exist. Request-safety is proven sound (m1).

The formalization of the Links programming language [32] is based on RPC calculus [29, 34] (m2) – an extension of lambda calculus – which models location awareness for stateful clients and stateless servers. The RPC calculus is transformed (s2) into a client program and a server program in the client/server calculus. The transformation is proved to be correct and complete (m1). Further, a location-aware calculus, which is the the theoretical foundation for the Links programming language, and a translation to RPC calculus is provided (p2). A simulation that proves that the behavior of the transformed program in the client/server calculus conforms to the behavior of the source program in location-aware calculus is left to future work.

Eliom [101] is formalized as an MT extension of core ML. The authors provide an operational semantics that formalizes the execution for an Eliom program (s1) and provide a translation (s2) separating an Eliom program into server and client ML programs. Besides subject reduction (p1), the authors prove the equivalence of the high level MT semantics with the semantics of the compiled client and server languages after splitting by simulation (p2). The simulation shows that, for any given source program, every reduction can be replayed in the transformed programs (m1). Eliom separates type universes for client and server, allowing the type system to track which values belong to which side. Eliom, however, leaves out interactive behavior, formalizing only the creation of a single page.

In ScalaLoci's formal semantics [132], the reduction relation is labeled with the distributed components on which a term is reduced (s1). The authors formulate soundness properties for the encoding of placement at the type level, e.g., that terms are reduced on the instances of the peers on which they are placed (p1). The type system is proven sound (m1).

Table 7. Distribution Topologies

Language	Distribution Topology			
	Client–Server	Client–Server + Database	Peer-to-Peer	Specifiable
Hop/Hop.js	●	·	·	·
Links	·	●	○[1]	·
Opa	·	●	·	·
StiP.js	●	·	·	·
Ur/Web	·	●	·	·
Eliom/Ocsigen	●	·	·	·
Gavial	●	·	·	·
AmbientTalk/R	·	·	●	·
ML5	●	·	·	○
ScalaLoci	·	·	·	●
WebSharper	●	·	·	·
Haste	●	·	·	·
Fun	●	·	·	·
Koka	●	·	·	○
Multi-Tier Calculus	●	·	·	○
Swift	●	·	·	·
Volta	●	·	·	·
J-Orchestra	·	·	·	●
Meteor	●	·	·	·
GWT	●	·	·	·

● Supported
○ Support conceptually possible, but not supported by the provided examples or the implementation
1 Client-to-client communication transparently through central server

Using the Koka language, it is possible to define a splitting function for the server and client parts of a program [73] based on Koka's ability to separate effectful computations (s3), which guarantees type soundness for the split programs (p1), e.g., an application can define a *client* effect consisting of DOM accesses and a *server* effect consisting of I/O operations (m2).

4.6 Distribution Topologies

Table 7 gives an overview over the distribution topologies supported by MT languages. The majority of MT approaches specifically targets **client–server** applications in the Web domain. Besides the client and the server tier, Links [32], Opa [102] and Ur/Web [28] also include language-level support for the **database** tier. Other MT languages require the use of additional libraries to access a database (e.g., Hop [110] or Eliom [101]).

Only few approaches target other distribution topologies: AmbientTalk [39] focuses on mobile ad hoc networks and allows services to be exported and discovered in a **peer-to-peer** manner, where peers are loosely coupled. ML5 [88] is an MT language which adopts the idea of *possible worlds* from models of modal logic to represent the different tiers in the distributed system. Worlds are used to assign resources to different tiers. Although this approach is potentially more general than the client–server model allowing for the definition of different tiers, the current compiler and runtime target web applications only. Similarly, in the MT calculus by Neubauer and Thiemann [90], locations are members of a set of location names that is not restricted to client and server. Their work, however, focuses on splitting code between a client and as server. Session-typed channels in Links [32] provide the illusion of client-to-client communication, but messages are routed through the server. In J-Orchestra [122], developers can define different interconnected network sites in a configuration file.

ScalaLoci [132] allows developers to **specify a distributed system's topology** by declaring types representing the different components and their relation. Thus, developers can define custom

architectural schemes (i.e., not only client–server) and specify various computing models (e.g., pipelines, rings, or master–worker schemes).

5 DISCUSSION AND OUTLOOK

In this section, we discuss open issues in MT programming and suggest future research directions.

5.1 Generic Distributed Systems

A significant limitation of most existing MT research languages (e.g., [28, 32, 95, 101, 102, 105, 110]) is that they do not address *generic* distributed systems but consider only the client–server architecture with clients of the same kind, mostly in the limited setting of web applications. Yet, many distributed systems require more complex architectures and configurations with different kinds of components – different types of clients, coordinators (e.g., in a master–worker scheme), backup nodes and logging services. The ScalaLoci MT language [132] contributes to this area with means to specify an architecture based on peer types, thus, supporting generic distributed systems, whose architecture can be defined by the developer.

It is likely that the lack of support for generic distributed architectures in most MT languages has limited the investigation of some aspects that are significant in distributed systems. For example, current MT languages consider only one level of consistency [130] (e.g., causal consistency), the one guaranteed – often implicitly – by the underlying communication system. However, in distributed applications, developers need to be able to choose among different levels of consistency and the safety/performance trade-off they offer. Further, existing MT languages do not provide dedicated language abstractions for designing fault-tolerant systems (e.g., actors' supervision trees). This state of things is motivated by the context where MT programming has been applied so far, the Web, where a permanent client failure cannot be recovered anyway.

5.2 Failures

In a distributed setting, including the Web, hosts may fail or disconnect without notice. In particular, for web applications, clients may close the browser at any point in time. To improve resiliency to faults, remote communication in MT languages is non-blocking, i.e., the program continues execution even when the remote communication channel is interrupted. Beyond that, some MT languages provide primitives that developers can use to detect disconnection, such as dedicated notification events, callbacks and exceptions.

Calling a service in Hop or GWT, for example, either invokes a success callback or a failure callback. In Stip.js, failure handling is defined via annotations (i.e., `@defineHandler` and `@useHandler`). Links' remote communication based on session types supports exception handling to deal with communication failures and disconnections. In ScalaLoci's event streams, failures are propagated downstream in a monadic fashion and developers can define failure handlers for upstream operators, similar to supervisors in actor systems. A special event signals the disconnection of a remote component. AmbientTalk provides fault-tolerant asynchronous message passing between distributed components. Messages sent to a disconnected component are buffered and delivered after the component reconnects. J-Orchestra allows developers to manually implement error handling by editing the code after splitting.

5.3 Programming in the Large

Current MT languages do not support dedicated modularization abstractions for programming in the large, such as module systems [74]. As a result, scalability for MT code bases is an open research topic, with the risk of severely hindering collaborative development and maintainability. There are two aspects to consider.

First, there is a technical challenge in the compilation process as the splitting into tier-specific code needs to be modular. For example, Ur/Web [28] supports a module system in the style of ML. However, Ur/Web does not feature separate compilation of modules since the language relies on whole-program analysis for slicing the application into client and server programs.

Second, an interesting research direction is to revisit existing modularization mechanisms to design them in synergy with MT abstractions, allowing the independent specification of placement and the combination of (multiple) modules through composition mechanisms (e.g., ML functors).

A notable exception to the lack of MT abstractions for programming in the large is the Eliom language [101]. In the context of Eliom, Radanne and Vouillon propose a module system [100] based on ML-style modules featuring functors to abstract over other modules. Eliom modules can contain client or server declarations (annotated as `%client` and `%server`). Mixed modules, defining both client and server code, span over the client–server boundary enabling software modularization along the modules direction as well as abstraction over the two tiers at the same time. Another example is the ScalaLoci [132] language for generic distributed systems, which supports a multitier module system [133] that uses *abstract peer types* to express the distributed architecture of the (sub)system encapsulated within each module. Developers use such abstract peer types to specify the placement of values at the type level and compose modules to combine the different (sub)system's architectures.

5.4 Controlled Experiments

Controlled experiments allow researchers to study the effect of languages on aspects such as development time, which cannot be easily inferred from analyzing program code. Unfortunately, we are not aware of empirical studies or controlled experiments that target MT programming. There are a number of aspects that can be measured, but a first step may entail an assessment of the effect of MT on program comprehension.

A promising option in this direction would be to consider exploratory studies such as interviews and the think-aloud approach [70, 71]. Also, MT programming combines functionalities that traditionally belong to different compilation units into into the same unit, which should be detectable with eye-tracking techniques, which have been successfully applied to understand how source code is inspected, debugged and comprehended by developers [20, 61, 65, 77, 125]. A different perspective is the effect of the MT paradigm on the cognitive models that developers build regarding software artifacts, or the the *bottom-up model* (or *situation model*) by Letovsky [75].

6 RELATED APPROACHES

In this section, we provide an overview of related research areas that influenced research on MT programming or share concepts with the MT paradigm.

PGAS Languages. Partitioned global address space languages (PGAS) [37] provide a high-level programming model for high-performance parallel execution. For example, X10 [26] parallelizes task execution based on a work-stealing scheduler, enabling programmers to write highly scalable code. Its programming model features explicit fork/join operations to make the cost of communication explicit. X10's sophisticated dependent type system [25] captures the *place* (the heap partition) a reference points to. Similar to MT languages, PGAS languages aim at reducing the boundaries between hosts, adopting a shared global address space to simplify development. The scope of PGAS languages, however, is very diverse – they focus on high performance computing in a dedicated cluster, while MT programming targets client–server architectures on the Internet.

Operator Placement. In contrast to explicit placement (e.g., via annotations), the operator placement problem consists of finding the *best* host on which each operator should be deployed in a

distributed system according to maximize a certain metric, such as throughput [35, 69] or load [27]. Methods in this field include the creation of overlay networks where operators are assigned to hosts via random selection [58], network modeling [98] and linear optimization to find the optimal solution to the constraint problem [23].

Software Architectures. Software architectures [47, 93] organize software systems into components and their connections as well as constraints on their interaction. Architecture description languages (ADL) [82] provide a mechanism for high-level specification and analysis of large software systems, for example, to guide architecture evolution. Yet, ADLs are often detached from implementation languages. ArchJava [3] paved the way for consolidating architecture specification and implementation in a single language. However, ArchJava does not specifically address distributed systems nor MT programming. Some approaches are at the intersection of MT and modeling languages: Hilda [139] is a web development environment for data-driven applications based on a high-level declarative language similar to UML which automatically partition MT software.

Actor Model. The Actor model, initially described by Hewitt [56] and available in popular implementations such as Erlang OTP [7] and Akka [2], encapsulates control and state into computation units that run concurrently and exchange messages asynchronously [1]. The decoupling offered by asynchronous communication and by the no-shared-memory approach enables implementing scalable and fault-tolerant systems. De Koster et al. [36] classify actor systems into four different variants: (i) the *classic actor model* allows for changing the current interface of an actor (i.e., the messages which an actor can process) by switching between different named behaviors, which handle different types of messages, (e.g., Rosette [123], Akka [2]), (ii) *active objects* define a single entry point with a fixed interface (e.g., SALSA [129], Orleans [21]), (iii) *process-based actors* are executed once and run until completion, supporting explicit receive operations during run time (e.g., Erlang [7], Scala Actor Library [53]) and (iv) *communicating event-loops* combine an object heap, a message queue and an event loop and support multiple interfaces simultaneously by defining different objects sharing the same message queue and event loop (e.g., E [86]). Actors, however, are a relatively low-level mechanism to program distributed systems, leaving programmers the manual work of breaking applications between message senders and message handlers. The survey by de Boer et al. [15] provides an overview of the current state of research on actors and active object languages.

Big Data Processing Systems. Part of the success of modern Big Data systems is due to a programming interface that – similar to MT programming – allows developers to define components that run on different hosts in the same compilation unit, with the framework adding communication and scheduling. This class of systems includes batch processing frameworks like Hadoop [38] and Spark [140], as well as stream processing systems like Flink [4] and Storm [118]. Since queries may process datasets that span multiple data centers and minimizing the traffic is crucial, approaches like Silos [68] offer abstractions that group nodes belonging to the same location so that the scheduler can minimize cross-data-center data transfer. Yet, in Big Data systems, the language semantics is visibly different, for example mutable shared variables are transformed in non-shared separated copies.

Language Integration for Database Queries. Properly integrating query languages into general-purpose languages is a long-standing research problem [8]. Compiling embedded queries into SQL was pioneered by the Kleisli system [138]. LINQ [124] is a language extension based on Kleisli's query compilation technique to uniformly access different data sources such as collections and relational databases. The Links [32] MT language also relies on this technique for providing access to the database tier. Recent approaches for embedding database queries, such as JReq [60], Ferry [51],

DBPL [109], Slick [115] or Quill [99], also follow a functional approach without object-relational mapping.

Multi-Stage Programming. Multi-stage programming splits program compilation into a number of stages, where the execution of one stage generates the code that is executed in the next stage. MetaML [120] and MetaOCaml [22] provide a quasi-quotation mechanism that is statically scoped to separate stages syntactically. Quoted expressions are not evaluated immediately but they generate code to be executed in the next stage. The Hop [110] MT language uses multi-stage programming to construct client code at the server side. Instead of using syntactic quotations, lightweight modular staging [108] employs a staging approach based on types, combining staged code fragments with strong guarantees on well-formedness and type soundness. Using lightweight modular staging with the Scala-virtualized modified Scala compiler [87], also enables overloading Scala language constructs such as loops and control structures.

Heterogeneous Computing. In heterogeneous computing, distributed systems consist of different kinds of processing devices, supporting different specialized processing features. The OpenCL standard [66] for implementing systems across heterogeneous platforms is rather low-level, requiring the programmer to be aware of the specific hardware, e.g., specifically redesigning serial algorithms into parallel ones. Approaches for improving programming heterogeneous systems include (i) compiler directives to offload computations to specialized processing units, independent of specific hardware characteristics [6], (ii) domain-specific embeddings for general-purpose languages [17, 72, 131] abstracting over low level details, such as compute kernel execution, and (iii) higher level programming models that provide primitives for a predefined set of operations [136].

Domain-Specific Languages. Several survey papers are available in the literature that provide an extensive overview of DSLs [83, 117, 126, 127]. Wile [137] provides a compendium of lessons learnt on developing domain-specific languages providing empirically derived guidelines for constructing and improving DSLs. So called *fourth generation* programming languages – following third generation hardware-independent general-purpose languages – are usually DSLs that provide higher levels of abstraction for a specific domain, such as data management, analysis and manipulation [44, 67].

Programming Languages for Distributed Systems. MT programming belongs to a long tradition of programming language design for distributed systems with influential distributed languages like Argus [79], Emerald [14], Distributed Oz [54, 128] and Dist-Orc [5]. More recently, there have been contributions to specific aspects in the design of programming languages that concern the support for distributed systems, such as cloud types to ensure eventual consistency [18], conflict-free replicated data types (CRDT) [114], language support for safe distribution of computations [85] and fault tolerance [84], as well as programming frameworks for mixed IoT/Cloud development, such as Ericsson's Calvin [94].

7 CONCLUSION

In this paper, we provide an overview of MT languages, a programming approach which combines the functionalities that belong to different tiers into the same compilation unit, delegating injection of communication code and generation of the deployment units to the compiler. We provide an overview of the existing solutions, discuss their positioning in the design space, including placement strategy, placement specification and granularity, degree of MT programming, communication abstractions, formalization, and supported architectures.

We hope that this paper can help researchers to orient themselves in the landscape of MT programming design as well as encourage future development of MT languages.

8 ACKNOWLEDGEMENTS

We would like to thank Simon Fowler, Manuel Serrano, Gabriel Radanne and Adam Chlipala for the feedback they provided on this manuscript, concerning (but not limited to) the Links language, the Hop and the Hop.js languages, the Eliom language and the Ur/Web language, respectively. This work has been co-funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – SFB 1053 – 210487104 – and – SFB 1119 – 236615297, by the DFG projects 322196540 and 383964710, by the LOEWE initiative (Hesse, Germany) within the emergenCITY centre and within the Software-Factory 4.0 project and by the German Federal Ministry of Education and Research and the Hessian State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

REFERENCES

[1] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA.
[2] Akka. 2009. http://akka.io/. Accessed 2020-05-05.
[3] Jonathan Aldrich, Craig Chambers, and David Notkin. 2002. ArchJava: Connecting Software Architecture to Implementation. In *Proceedings of the 24th International Conference on Software Engineering* (Orlando, FL, USA) *(ICSE '02)*. ACM, New York, NY, USA, 187–197. https://doi.org/10.1145/581339.581365
[4] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. 2014. The Stratosphere Platform for Big Data Analytics. *The VLDB Journal* 23, 6 (Dec. 2014), 939–964. https://doi.org/10.1007/s00778-014-0357-y
[5] Musab AlTurki and José Meseguer. 2010. Dist-Orc: A Rewriting-based Distributed Implementation of Orc with Formal Analysis. In *Proceedings First International Workshop on Rewriting Techniques for Real-Time Systems* (Longyearbyen, Norway) *(RTRTS '10)*. 26–45. https://doi.org/10.4204/EPTCS.36.2
[6] José M. Andión, Manuel Arenaz, François Bodin, Gabriel Rodríguez, and Juan Touriño. 2016. Locality-Aware Automatic Parallelization for GPGPU with OpenHMPP Directives. *International Journal of Parallel Programming* 44, 3 (June 2016), 620–643. https://doi.org/10.1007/s10766-015-0362-9
[7] Joe Armstrong. 2010. Erlang. *Commun. ACM* 53, 9 (Sept. 2010), 68–75. https://doi.org/10.1145/1810891.1810910
[8] Malcolm P. Atkinson and O. Peter Buneman. 1987. Types and Persistence in Database Programming Languages. *Comput. Surveys* 19, 2 (June 1987), 105–170. https://doi.org/10.1145/62070.45066
[9] Vincent Balat. 2006. Ocsigen: Typing Web Interaction with Objective Caml. In *Proceedings of the 2006 Workshop on ML* (Portland, OR, USA) *(ML '06)*. ACM, New York, NY, USA, 84–94. https://doi.org/10.1145/1159876.1159889
[10] Ioannis G. Baltopoulos and Andrew D. Gordon. 2009. Secure Compilation of a Multi-Tier Web Language. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation* (Savannah, GA, USA) *(TLDI '09)*. ACM, New York, NY, USA, 27–38. https://doi.org/10.1145/1481861.1481866
[11] Gérard Berry and Georges Gonthier. 1992. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming* 19, 2 (Nov. 1992), 87–152. https://doi.org/10.1016/0167-6423(92)90005-V
[12] Gérard Berry, Cyprien Nicolas, and Manuel Serrano. 2011. HipHop: A Synchronous Reactive Extension for Hop. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients* (Portland, OR, USA) *(PLASTIC '11)*. ACM, New York, NY, USA, 49–56. https://doi.org/10.1145/2093328.2093337
[13] Joel Bjornson, Anton Tayanovskyy, and Adam Granicz. 2010. Composing Reactive GUIs in F# Using WebSharper. In *Proceedings of the 22nd International Conference on Implementation and Application of Functional Languages* (Alphen aan den Rijn, The Netherlands) *(IFL '10)*. Springer-Verlag, Berlin, Heidelberg, 203–216. https://doi.org/10.1007/978-3-642-24276-2_13
[14] Andrew P. Black, Norman C. Hutchinson, Eric Jul, and Henry M. Levy. 2007. The Development of the Emerald Programming Language. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages* (San Diego, California) *(HOPL III)*. ACM, New York, NY, USA, 11:1–11:51. https://doi.org/10.1145/1238844.1238855
[15] Frank de Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and et al. 2017. A Survey of Active Object Languages. *Comput. Surveys* 50, 5, Article 76 (Oct. 2017), 39 pages. https://doi.org/10.1145/3122848
[16] Gérard Boudol, Zhengqin Luo, Tamara Rezk, and Manuel Serrano. 2012. Reasoning about Web Applications: An Operational Semantics for Hop. *ACM Transactions on Programming Languages and Systems* 34, 2, Article 10 (June

2012), 40 pages. https://doi.org/10.1145/2220365.2220369

[17] Jens Breitbart. 2009. CuPP – A Framework for Easy CUDA Integration. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing* (Rome, Italy) *(IPDPS '09)*. IEEE Computer Society, Washington, DC, USA, 8. https://doi.org/10.1109/IPDPS.2009.5160937

[18] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. 2012. Cloud Types for Eventual Consistency. In *Proceedings of the 26th European Conference on Object-Oriented Programming* (Beijing, China) *(ECOOP '12)*. Springer-Verlag, Berlin, Heidelberg, 283–307. https://doi.org/10.1007/978-3-642-31057-7_14

[19] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. *Pattern-Oriented Software Architecture: A System of Patterns*. Vol. 1. John Wiley & Sons.

[20] Teresa Busjahn, Carsten Schulte, Bonita Sharif, Simon, Andrew Begel, Michael Hansen, Roman Bednarik, Paul Orlov, Petri Ihantola, Galina Shchekotova, and Maria Antropova. 2014. Eye Tracking in Computing Education. In *Proceedings of the Tenth Annual Conference on International Computing Education Research* (Glasgow, Scotland, United Kingdom) *(ICER '14)*. ACM, New York, NY, USA, 3–10. https://doi.org/10.1145/2632320.2632344

[21] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. 2011. Orleans: Cloud Computing for Everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (Cascais, Portugal) *(SOCC '11)*. ACM, New York, NY, USA, Article 16, 14 pages. https://doi.org/10.1145/2038916.2038932

[22] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. 2003. Implementing Multi-Stage Languages Using ASTs, Gensym, and Reflection. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering* (Erfurt, Germany) *(GPCE '03)*. Springer-Verlag, Berlin, Heidelberg, 57–76. https://doi.org/doi.org/10.1007/978-3-540-39815-8_4

[23] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. 2016. Optimal Operator Placement for Distributed Stream Processing Applications. In *Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems* (Irvine, CA, USA) *(DEBS '16)*. ACM, New York, NY, USA, 69–80. https://doi.org/10.1145/2933267.2933312

[24] Andoni Lombide Carreton, Stijn Mostinckx, Tom Van Cutsem, and Wolfgang De Meuter. 2010. Loosely-coupled Distributed Reactive Programming in Mobile Ad Hoc Networks. In *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns* (Málaga, Spain) *(TOOLS '10)*. Springer-Verlag, Berlin, Heidelberg, 41–60. https://doi.org/10.1007/978-3-642-13953-6_3

[25] Satish Chandra, Vijay Saraswat, Vivek Sarkar, and Rastislav Bodik. 2008. Type Inference for Locality Analysis of Distributed Data Structures. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Salt Lake City, UT, USA) *(PPoPP '08)*. ACM, New York, NY, USA, 11–22. https://doi.org/10.1145/1345206.1345211

[26] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. ACM, New York, NY, USA, 519–538. https://doi.org/10.1145/1094811.1094852

[27] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Çetintemel, Ying Xing, and Stan Zdonik. 2003. Scalable Distributed Stream Processing. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research* (Asilomar, CA, USA) *(CIDR '03)*. http://www-db.cs.wisc.edu/cidr/cidr2003/program/p23.pdf Accessed 2020-05-05.

[28] Adam Chlipala. 2015. Ur/Web: A Simple Model for Programming the Web. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. ACM, New York, NY, USA, 153–165. https://doi.org/10.1145/2676726.2677004

[29] Kwanghoon Choi and Byeong-Mo Chang. 2019. A Theory of RPC Calculi for Client–Server Model. *Journal of Functional Programming* 29 (2019). https://doi.org/10.1017/S0956796819000029

[30] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. 2007. Secure Web Applications via Automatic Partitioning. *ACM SIGOPS Operating Systems Review* 41, 6 (Oct. 2007), 31–44. https://doi.org/10.1145/1323293.1294265

[31] Stephen Chong, K. Vikram, and Andrew C. Myers. 2007. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *Proceedings of 16th USENIX Security Symposium* (Boston, MA, USA) *(SS '07)*. USENIX Association, Berkeley, CA, USA, Article 1, 16 pages. http://usenix.org/events/sec07/tech/full_papers/chong/chong.pdf Accessed 2020-05-05.

[32] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2006. Links: Web Programming without Tiers. In *Proceedings of the 5th International Conference on Formal Methods for Components and Objects* (Amsterdam, The Netherlands) *(FMCO '06)*. Springer-Verlag, Berlin, Heidelberg, 266–296. https://doi.org/10.1007/978-3-540-74792-5_12

[33] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2008. The Essence of Form Abstraction. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems* (Bangalore, India) *(APLAS '08)*. Springer-Verlag, Berlin, Heidelberg, 205–220. https://doi.org/10.1007/978-3-540-89330-1_15

[34] Ezra E. K. Cooper and Philip Wadler. 2009. The RPC Calculus. In *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming* (Coimbra, Portugal) *(PPDP '09)*. ACM, New York, NY, USA, 231–242. https://doi.org/10.1145/1599410.1599439

[35] Gianpaolo Cugola and Alessandro Margara. 2013. Deployment Strategies for Distributed Complex Event Processing. *Computing* 95, 2 (Feb. 2013), 129–156. https://doi.org/10.1007/s00607-012-0217-9

[36] Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 2016. 43 Years of Actors: A Taxonomy of Actor Models and Their Key Properties. In *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control* (Amsterdam, Netherlands) *(AGERE '16)*. ACM, New York, NY, USA, 31–40. https://doi.org/10.1145/3001886.3001890

[37] Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. 2015. Partitioned Global Address Space Languages. *Comput. Surveys* 47, 4, Article 62 (May 2015), 27 pages. https://doi.org/10.1145/2716320

[38] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[39] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. 2006. Ambient-Oriented Programming in Ambienttalk. In *Proceedings of the 20th European Conference on Object-Oriented Programming* (Nantes, France) *(ECOOP '06)*. Springer-Verlag, Berlin, Heidelberg, 230–254. https://doi.org/10.1007/11785477_16

[40] Gwenaël Delaval, Alain Girault, and Marc Pouzet. 2008. A Type System for the Automatic Distribution of Higher-Order Synchronous Dataflow Programs. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (Tucson, AZ, USA) *(LCTES '08)*. ACM, New York, NY, USA, 101–110. https://doi.org/10.1145/1375657.1375672

[41] Anton Ekblad and Koen Claessen. 2014. A Seamless, Client-centric Programming Model for Type Safe Web Applications. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell* (Gothenburg, Sweden) *(Haskell '14)*. ACM, New York, NY, USA, 79–89. https://doi.org/10.1145/2633357.2633367

[42] Stefan Fehrenbach and James Cheney. 2019. Language-Integrated Provenance by Trace Analysis. In *Proceedings of the 17th ACM SIGPLAN International Symposium on Database Programming Languages* (Phoenix, AZ, USA) *(DBPL '19)*. ACM, New York, NY, USA, 74–84. https://doi.org/10.1145/3315507.3330198

[43] Haxe Foundation. 2005. Haxe cross-platform toolkit. http://haxe.org. Accessed 2020-05-05.

[44] Martin Fowler. 2010. *Domain Specific Languages* (1st ed.). Addison-Wesley Professional.

[45] Simon Fowler. 2020. Model-View-Update-Communicate: Session Types meet the Elm Architecture. In *Proceedings of the 34rd European Conference on Object-Oriented Programming (ECOOP 2019)* (Berlin, Germany) *(Leibniz International Proceedings in Informatics (LIPIcs))*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 29.

[46] Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. 2019. Exceptional Asynchronous Session Types: Session Types without Tiers. *Proceedings of the ACM on Programming Languages* 3, POPL, Article 28 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290341

[47] David Garlan and Mary Shaw. 1994. *An Introduction to Software Architecture.* Technical Report. Pittsburgh, PA, USA. http://www.cs.cmu.edu/afs/cs/project/vit/ftp/pdf/intro_softarch.pdf Accessed 2020-05-05.

[48] David Gelernter. 1985. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems* 7, 1 (Jan. 1985), 80–112. https://doi.org/10.1145/2363.2433

[49] Danny M. Groenewegen, Zef Hemel, Lennart C. L. Kats, and Eelco Visser. 2008. WebDSL: A Domain-Specific Language for Dynamic Web Applications. In *Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications* (Nashville, TN, USA) *(OOPSLA Companion '08)*. ACM, New York, NY, USA, 779–780. https://doi.org/10.1145/1449814.1449858

[50] Object Management Group. 1993. *The Common Object Request Broker: Architecture and Specification.* Wiley-QED.

[51] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. 2009. Ferry – Database-Supported Program Execution. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (Providence, RI, USA) *(SIGMOD '09)*. ACM, New York, NY, USA, 1063–1066. https://doi.org/10.1145/1559845.1559982

[52] Arjun Guha, Jean-Baptiste Jeannin, Rachit Nigam, Jane Tangen, and Rian Shambaugh. 2017. Fission: Secure Dynamic Code-Splitting for JavaScript. In *Proceedings of the 2nd Summit on Advances in Programming Languages (SNAPL 2017)* (Asilomar, CA, USA) *(Leibniz International Proceedings in Informatics (LIPIcs), Vol. 71)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 5:1–5:13. https://doi.org/10.4230/LIPIcs.SNAPL.2017.5

[53] Philipp Haller and Martin Odersky. 2009. Scala Actors: Unifying Thread-Based and Event-Based Programming. *Theoretical Computer Science* 410, 2–3 (Feb. 2009), 202–220. https://doi.org/10.1016/j.tcs.2008.09.019

[54] Seif Haridi, Peter Van Roy, and Gert Smolka. 1997. An Overview of the Design of Distributed Oz. In *Proceedings of the Second International Symposium on Parallel Symbolic Computation* (Maui, HI, USA) *(PASCO '97)*. ACM, New York, NY, USA, 176–187. https://doi.org/10.1145/266670.266726

[55] Zef Hemel and Eelco Visser. 2011. Declaratively Programming the Mobile Web with Mobl. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, OR, USA) *(OOPSLA '11)*. ACM, New York, NY, USA, 695–712. https://doi.org/10.1145/2048066.2048121

[56] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular Actor Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence* (Stanford, CA, USA) *(IJCAI '73)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 235–245. http://ijcai.org/Proceedings/73/Papers/027B.pdf Accessed 2020-05-05.

[57] Daniel Hillerström, Sam Lindley, Robert Atkey, and KC Sivaramakrishnan. 2017. Continuation Passing Style for Effect Handlers. In *Proceedings of the 2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017)* (Oxford, UK) *(Leibniz International Proceedings in Informatics (LIPIcs), Vol. 84)*, Dale Miller (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 18:1–18:19. https://doi.org/10.4230/LIPIcs.FSCD.2017.18

[58] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. 2003. Querying the Internet with PIER. In *Proceedings of the 29th International Conference on Very Large Data Bases* (Berlin, Germany) *(VLDB '03)*. VLDB Endowment, 321–332. https://doi.org/10.1016/B978-012722442-8/50036-7

[59] Galen C. Hunt and Michael L. Scott. 1999. The Coign Automatic Distributed Partitioning System. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (New Orleans, LA, USA) *(OSDI '99)*. USENIX Association, Berkeley, CA, USA, 187–200. http://usenix.org/events/osdi99/full_papers/hunt/hunt.pdf Accessed 2020-05-05.

[60] Ming-Yee Iu, Emmanuel Cecchet, and Willy Zwaenepoel. 2010. JReq: Database Queries in Imperative Languages. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction* (Paphos, Cyprus) *(CC/ETAPS '10)*. Springer-Verlag, Berlin, Heidelberg, 84–103. https://doi.org/10.1007/978-3-642-11970-5_6

[61] Ahmad Jbara and Dror G. Feitelson. 2015. How Programmers Read Regular Code: A Controlled Experiment Using Eye Tracking. In *Proceedings of the 23rd IEEE International Conference on Program Comprehension* (Florence, Italy) *(ICPC '15)*. IEEE Press, Piscataway, NJ, USA, 244–254. https://doi.org/10.1109/ICPC.2015.35

[62] JetBrains. 2009. Kotlin programming language. http://kotlinlang.org. Accessed 2020-05-05.

[63] Samuel C. Kendall, Jim Waldo, Ann Wollrath, and Geoff Wyant. 1994. *A Note on Distributed Computing*. Technical Report. Sun Microsystems, Inc., Mountain View, CA, USA.

[64] Federico Kereki. 2010. *Essential GWT: Building for the Web with Google Web Toolkit 2* (1st ed.). Addison-Wesley Professional.

[65] Katja Kevic, Braden M. Walters, Timothy R. Shaffer, Bonita Sharif, David C. Shepherd, and Thomas Fritz. 2015. Tracing Software Developers' Eyes and Interactions for Change Tasks. In *Proceedings of the the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE '15)*. ACM, New York, NY, USA, 202–213. https://doi.org/10.1145/2786805.2786864

[66] Khronos OpenCL Working Group. 2009. The OpenCL Specification. In *Proceedings of the 2009 IEEE Hot Chips 21 Symposium* (Stanford, CA, USA) *(HCS '09)*. 314. https://doi.org/10.1109/HOTCHIPS.2009.7478342

[67] Robert Klepper and Douglas Bock. 1995. Third and Fourth Generation Language Productivity Differences. *Commun. ACM* 38, 9 (Sept. 1995), 69–79. https://doi.org/10.1145/223248.223268

[68] Konstantinos Kloudas, Margarida Mamede, Nuno Preguiça, and Rodrigo Rodrigues. 2015. Pixida: Optimizing Data Parallel Jobs in Wide-Area Data Analytics. *Proceedings of the VLDB Endowment* 9, 2 (Oct. 2015), 72–83. https://doi.org/10.14778/2850578.2850582

[69] Geetika T. Lakshmanan, Ying Li, and Rob Strom. 2008. Placement Strategies for Internet-Scale Data Stream Systems. *IEEE Internet Computing* 12, 6 (Nov. 2008), 50–60. https://doi.org/10.1109/MIC.2008.129

[70] Thomas D. LaToza, David Garlan, James D. Herbsleb, and Brad A. Myers. 2007. Program Comprehension as Fact Finding. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (Dubrovnik, Croatia) *(ESEC/FSE '07)*. ACM, New York, NY, USA, 361–370. https://doi.org/10.1145/1287624.1287675

[71] Thomas D. LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining Mental Models: A Study of Developer Work Habits. In *Proceedings of the 28th International Conference on Software Engineering* (Shanghai, China) *(ICSE '06)*. ACM, New York, NY, USA, 492–501. https://doi.org/10.1145/1134285.1134355

[72] Orion Sky Lawlor. 2011. Embedding OpenCL in C++ for Expressive GPU Programming. In *Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing* (Tucson, AZ, USA) *(WOLFHPC '11)*. http://hpc.pnl.gov/conf/wolfhpc/2011/papers/L2011.pdf Accessed 2020-05-05.

[73] Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. (2014), 100–126. https://doi.org/10.4204/EPTCS.153.8

[74] Xavier Leroy. 2000. A Modular Module System. *Journal of Functional Programming* 10, 3 (May 2000), 269–303. https://doi.org/10.1017/S0956796800003683

[75] Stanley Letovsky. 1986. Cognitive Processes in Program Comprehension. In *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers* (Washington, D.C., USA). Ablex Publishing Corp., Norwood, NJ, USA, 58–79.

[76] Haoyi Li. 2012. ScalaTags. http://www.lihaoyi.com/scalatags/. Accessed 2020-05-05.

[77] Yu-Tzu Lin, Cheng-Chih Wu, Ting-Yun Hou, Yu-Chih Lin, Fang-Ying Yang, and Chia-Hu Chang. 2016. Tracking Students' Cognitive Processes During Program Debugging – An Eye-Movement Approach. *IEEE Transactions on Education* 59, 3 (2016), 175–186. https://doi.org/10.1109/TE.2015.2487341

[78] Sam Lindley and J. Garrett Morris. 2017. Lightweight Functional Session Types. In *Behavioural Types: From Theory to Tools*, Simon Gay and António Ravara (Eds.). River Publishers, Chapter 12. https://doi.org/10.13052/rp-9788793519817

[79] Barbara Liskov. 1988. Distributed Programming in Argus. *Commun. ACM* 31, 3 (March 1988), 300–312. https://doi.org/10.1145/42392.42399

[80] Zhengqin Luo, Tamara Rezk, and Manuel Serrano. 2011. Automated Code Injection Prevention for Web Applications. In *Proceedings of the 2011 International Conference on Theory of Security and Applications* (Saarbrücken, Germany) *(TOSCA '11)*. Springer-Verlag, Berlin, Heidelberg, 186–204. https://doi.org/10.1007/978-3-642-27375-9_11

[81] Dragos Manolescu, Brian Beckman, and Benjamin Livshits. 2008. Volta: Developing Distributed Applications by Recompiling. *IEEE Software* 25, 5 (Sept. 2008), 53–59. https://doi.org/10.1109/MS.2008.131

[82] Nenad Medvidovic and Richard N. Taylor. 2000. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering* 26, 1 (Jan. 2000), 70–93. https://doi.org/10.1109/32.825767

[83] Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. When and How to Develop Domain-Specific Languages. *Comput. Surveys* 37, 4 (Dec. 2005), 316–344. https://doi.org/10.1145/1118890.1118892

[84] Heather Miller, Philipp Haller, Normen Müller, and Jocelyn Boullier. 2016. Function Passing: A Model for Typed, Distributed Functional Programming. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Amsterdam, Netherlands) *(Onward! 2016)*. ACM, New York, NY, USA, 82–97. https://doi.org/10.1145/2986012.2986014

[85] Heather Miller, Philipp Haller, and Martin Odersky. 2014. Spores: A Type-Based Foundation for Closures in the Age of Concurrency and Distribution. In *Proceedings of the 28th European Conference on Object-Oriented Programming* (Uppsala, Sweden) *(ECOOP '14)*. Springer-Verlag, Berlin, Heidelberg, 308–333. https://doi.org/10.1007/978-3-662-44202-9_13

[86] Mark S. Miller, E. Dean Tribble, and Jonathan Shapiro. 2005. Concurrency Among Strangers: Programming in E as Plan Coordination. In *Proceedings of the 1st International Conference on Trustworthy Global Computing* (Edinburgh, UK) *(TGC '05)*. Springer-Verlag, Berlin, Heidelberg, 195–229. https://doi.org/10.1007/11580850_12

[87] Adriaan Moors, Tiark Rompf, Philipp Haller, and Martin Odersky. 2012. Scala-Virtualized. In *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation* (Philadelphia, PA, USA) *(PEPM '12)*. ACM, New York, NY, USA, 117–120. https://doi.org/10.1145/2103746.2103769

[88] Tom Murphy, VII, Karl Crary, and Robert Harper. 2007. Type-safe Distributed Programming with ML5. In *Proceedings of the 3rd Conference on Trustworthy Global Computing* (Sophia-Antipolis, France) *(TGC '07)*. Springer-Verlag, Berlin, Heidelberg, 108–123. https://doi.org/10.1007/978-3-540-78663-4_9

[89] Tim Nelson, Andrew D. Ferguson, Michael J. G. Scheer, and Shriram Krishnamurthi. 2014. Tierless Programming and Reasoning for Software-Defined Networks. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Seattle, WA, USA) *(NSDI '14)*. USENIX Association, Berkeley, CA, USA, 519–531. http://usenix.org/system/files/conference/nsdi14/nsdi14-paper-nelson.pdf Accessed 2020-05-05.

[90] Matthias Neubauer and Peter Thiemann. 2005. From Sequential Programs to Multi-Tier Applications by Program Transformation. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, CA, USA) *(POPL '05)*. ACM, New York, NY, USA, 221–232. https://doi.org/10.1145/1040305.1040324

[91] Matthias Neubauer and Peter Thiemann. 2008. Placement Inference for a Client-Server Calculus. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming, Part II* (Reykjavik, Iceland) *(ICALP '08)*. Springer-Verlag, Berlin, Heidelberg, 75–86. https://doi.org/10.1007/978-3-540-70583-3_7

[92] David Park. 1981. Concurrency and Automata on Infinite Sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science* (Karlsruhe, Germany). Springer-Verlag, Berlin, Heidelberg, 167–183. https://doi.org/10.1007/BFb0017309

[93] Dewayne E. Perry and Alexander L. Wolf. 1992. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes* 17, 4 (Oct. 1992), 40–52. https://doi.org/10.1145/141874.141884

[94] Per Persson and Ola Angelsmark. 2015. Calvin – Merging Cloud and IoT. *Procedia Computer Science* 52, The 6th International Conference on Ambient Systems, Networks and Technologies, the 5th International Conference on Sustainable Energy Information Technology (2015), 210–217. https://doi.org/10.1016/j.procs.2015.05.059

[95] Laure Philips, Joeri De Koster, Wolfgang De Meuter, and Coen De Roover. 2018. Search-based Tier Assignment for Optimising Offline Availability in Multi-tier Web Applications. *The Art, Science, and Engineering of Programming* 2, 2 (Dec. 2018), 3:1–3:29. https://doi.org/10.22152/programming-journal.org/2018/2/3

[96] Laure Philips, Coen De Roover, Tom Van Cutsem, and Wolfgang De Meuter. 2014. Towards Tierless Web Development without Tierless Languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Portland, OR, USA) *(Onward! 2014)*. ACM, New York, NY, USA, 69–81. https://doi.org/10.1145/2661136.2661146

[97] Laure Philips, Coen De Roover, Tom Van Cutsem, and Wolfgang De Meuter. 2014. Towards Tierless Web Development Without Tierless Languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014)*. ACM, New York, NY, USA, 69–81. https://doi.org/10.1145/2661136.2661146

[98] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. 2006. Network-Aware Operator Placement for Stream-Processing Systems. In *Proceedings of the 22nd International Conference on Data Engineering* (Atlanta, GA, USA) *(ICDE '06)*. IEEE Computer Society, Washington, DC, USA, 49–60. https://doi.org/10.1109/ICDE.2006.105

[99] Quill. 2015. http://getquill.io/. Accessed 2020-05-05.

[100] Gabriel Radanne and Jérôme Vouillon. 2018. Tierless Web Programming in the Large. In *Companion Proceedings of the The Web Conference 2018* (Lyon, France) *(WWW '18)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 681–689. https://doi.org/10.1145/3184558.3185953

[101] Gabriel Radanne, Jérôme Vouillon, and Vincent Balat. 2016. Eliom: A core ML language for Tierless Web Programming. In *Proceedings of the 14th Asian Symposium on Programming Languages and Systems* (Hanoi, Vietnam) *(APLAS '16)*. Springer-Verlag, Berlin, Heidelberg, 377–397. https://doi.org/10.1007/978-3-319-47958-3_20

[102] David Rajchenbach-Teller and Franois-Régis Sinot. 2010. Opa: Language Support for a Sane, Safe and Secure Web. In *Proceedings of the OWASP AppSec Research* (Stockholm, Sweden). http://owasp.org/www-pdf-archive/OWASP_AppSec_Research_2010_OPA_by_Rajchenbach-Teller.pdf Accessed 2020-05-05.

[103] Jan S. Rellermeyer, Gustavo Alonso, and Timothy Roscoe. 2007. R-OSGi: Distributed Applications through Software Modularization. In *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware* (Newport Beach, CA, USA) *(Middleware '07)*. Springer-Verlag, Berlin, Heidelberg, 1–20. https://doi.org/10.1007/978-3-540-76778-7_1

[104] Bob Reynders, Dominique Devriese, and Frank Piessens. 2014. Multi-Tier Functional Reactive Programming for the Web. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Portland, OR, USA) *(Onward! 2014)*. ACM, New York, NY, USA, 55–68. https://doi.org/10.1145/2661136.2661140

[105] Bob Reynders, Frank Piessens, and Dominique Devriese. 2020. Gavial: Programming the Web with Multi-Tier FRP. *The Art, Science, and Engineering of Programming* 4, 3 (Feb. 2020), 6:1–6:32. https://doi.org/10.22152/programming-journal.org/2020/4/6

[106] Julien Richard-Foy, Olivier Barais, and Jean-Marc Jézéquel. 2013. Efficient High-Level Abstractions for Web Programming. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences* (Indianapolis, IN, USA) *(GPCE '13)*. ACM, New York, NY, USA, 53–60. https://doi.org/10.1145/2517208.2517227

[107] Java RMI. 1999. *Java Remote Method Invocation – Distributed Computing for Java.* Technical Report. Sun Microsystems, Inc., Mountain View, CA, USA. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html Accessed 2020-05-05.

[108] Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proceedings of the 9th International Conference on Generative Programming and Component Engineering* (Eindhoven, The Netherlands) *(GPCE '10)*. ACM, New York, NY, USA, 127–136. https://doi.org/10.1145/1868294.1868314

[109] Joachim W. Schmidt and Florian Matthes. 1994. The DBPL Project: Advances in Modular Database Programming. *Information Systems* 19, 2 (March 1994), 121–140. https://doi.org/10.1016/0306-4379(94)90007-8

[110] Manuel Serrano, Erick Gallesio, and Florian Loitsch. 2006. Hop, A Language for Programming the Web 2.0. In *Companion to the 21th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Portland, OR, USA) *(OOPSLA Companion '06)*. ACM, New York, NY, USA.

[111] Manuel Serrano and Vincent Prunet. 2016. A Glimpse of Hopjs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP '16)*. ACM, New York, NY, USA, 180–192. https://doi.org/10.1145/2951913.2951916

[112] Manuel Serrano and Christian Queinnec. 2010. A Multi-Tier Semantics for Hop. *Higher-Order and Symbolic Computation* 23, 4 (Nov. 2010), 409–431. https://doi.org/10.1007/s10990-010-9061-9

[113] Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. 2005. Acute: High-Level Programming Language Design for Distributed Computation. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming* (Tallinn, Estonia) *(ICFP '05)*. ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/1086365.1086370

[114] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems* (Grenoble, France) *(SSS '11)*. Springer-Verlag, Berlin, Heidelberg, 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

[115] Slick. 2014. http://scala-slick.org/. Accessed 2020-05-05.

[116] Elliot Soloway and Kate Ehrlich. 1984. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering* 10, 5 (Sept. 1984), 595–609. https://doi.org/10.1109/TSE.1984.5010283

[117] Diomidis Spinellis. 2001. Notable Design Patterns for Domain-Specific Languages. *Journal of Systems and Software* 56, 1 (Feb. 2001), 91–99. https://doi.org/10.1016/S0164-1212(00)00089-3

[118] Storm. 2011. http://storm.apache.org/. Accessed 2020-05-05.

[119] Isaac Strack. 2012. *Getting Started with Meteor.js JavaScript Framework* (1st ed.). Packt Publishing.

[120] Walid Taha and Tim Sheard. 1997. Multi-Stage Programming with Explicit Annotations. (1997), 203–217. https://doi.org/10.1145/258993.259019

[121] John A. Thywissen, Arthur Michener Peters, and William R. Cook. 2016. Implicitly Distributing Pervasively Concurrent Programs: Extended Abstract. In *Proceedings of the 1st Workshop on Programming Models and Languages for Distributed Computing* (Rome, Italy) *(PMLDC '16)*. ACM, New York, NY, USA, Article 1, 4 pages. https://doi.org/10.1145/2957319.2957370

[122] Eli Tilevich and Yannis Smaragdakis. 2002. J-Orchestra: Automatic Java Application Partitioning. In *Proceedings of the 16th European Conference on Object-Oriented Programming* (London, UK) *(ECOOP '02)*, Boris Magnusson (Ed.). Springer-Verlag, Berlin, Heidelberg, 178–204. https://doi.org/10.1007/3-540-47993-7_8

[123] Chris Tomlinson, Won Kim, Mark Scheevel, Vineet Singh, Becky Will, and Gul Agha. 1988. Rosette: An Object-Oriented Concurrent Systems Architecture. In *Proceedings of the 1988 ACM SIGPLAN Workshop on Object-Based Concurrent Programming* (San Diego, CA, USA) *(OOPSLA/ECOOP Companion '88)*. ACM, New York, NY, USA, 91–93. https://doi.org/10.1145/67386.67410

[124] Mads Torgersen. 2007. Querying in C#: How Language Integrated Query (LINQ) Works. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion* (Montreal, Quebec, Canada) *(OOPSLA Companion '07)*. ACM, New York, NY, USA, 852–853. https://doi.org/10.1145/1297846.1297922

[125] Rachel Turner, Michael Falcone, Bonita Sharif, and Alina Lazar. 2014. An Eye-Tracking Study Assessing the Comprehension of C++ and Python Source Code. In *Proceedings of the Symposium on Eye Tracking Research and Applications* (Safety Harbor, Florida) *(ETRA '14)*. ACM, New York, NY, USA, 231–234. https://doi.org/10.1145/2578153.2578218

[126] Arie van Deursen and Paul Klint. 1998. Little Languages: Little Maintenance? *Journal of Software Maintenance: Research and Practice* 10, 2 (1998), 75–92. https://doi.org/10.1002/(SICI)1096-908X(199803/04)10:2<75::AID-SMR168>3.0.CO;2-5

[127] Arie van Deursen, Paul Klint, and Joost Visser. 2000. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices* 35, 6 (June 2000), 26–36. https://doi.org/10.1145/352029.352035

[128] Peter Van Roy, Seif Haridi, Per Brand, Gert Smolka, Michael Mehl, and Ralf Scheidhauer. 1997. Mobile Objects in Distributed Oz. *ACM Transactions on Programming Languages and Systems* 19, 5 (Sept. 1997), 804–851. https://doi.org/10.1145/265943.265972

[129] Carlos Varela and Gul Agha. 2001. Programming Dynamically Reconfigurable Open Systems with SALSA. *ACM SIGPLAN Notices* 36, 12 (Dec. 2001), 20–34. https://doi.org/10.1145/583960.583964

[130] Paolo Viotti and Marko Vukolić. 2016. Consistency in Non-Transactional Distributed Storage Systems. *Comput. Surveys* 49, 1, Article 19 (June 2016), 34 pages. https://doi.org/10.1145/2926965

[131] Moisés Viñas, Zeki Bozkus, and Basilio B. Fraguela. 2013. Exploiting Heterogeneous Parallelism with the Heterogeneous Programming Library. *J. Parallel and Distrib. Comput.* 73, 12 (Dec. 2013), 1627–1638. https://doi.org/10.1016/j.jpdc.2013.07.013

[132] Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. 2018. Distributed System Development with ScalaLoci. *Proceedings of the ACM on Programming Languages* 2, OOPSLA, Article 129 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276499

[133] Pascal Weisenburger and Guido Salvaneschi. 2019. Multitier Modules. In *Proceedings of the 33rd European Conference on Object-Oriented Programming (ECOOP 2019)* (London, UK) *(Leibniz International Proceedings in Informatics (LIPIcs), Vol. 134)*, Alastair F. Donaldson (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 3:1–3:29. https://doi.org/10.4230/LIPIcs.ECOOP.2019.3

[134] Mark Weiser. 1981. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering* (San Diego, CA, USA) *(ICSE '81)*. IEEE Press, Piscataway, NJ, USA, 439–449.

[135] Marcus Westin. 2010. Fun: A programming language for the realtime web. http://marcuswest.in/essays/fun-intro/. Accessed 2020-05-05.

[136] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. 2012. OpenACC – First Experiences with Real-World Applications. In *Proceedings of the 18th International Conference on Parallel Processing* (Rhodes Island, Greece) *(Euro-Par '12)*. Springer-Verlag, Berlin, Heidelberg, 859–870. https://doi.org/10.1007/978-3-642-32820-6_85

[137] David Wile. 2004. Lessons Learned from Real DSL Experiments. *Science of Computer Programming* 51, 3 (June 2004), 265–290. https://doi.org/10.1016/j.scico.2003.12.006

[138] Limsoon Wong. 2000. Kleisli, a Functional Query System. *Journal of Functional Programming* 10, 1 (Jan. 2000), 19–56. https://doi.org/10.1017/S0956796899003585

[139] Fan Yang, Nitin Gupta, Nicholas Gerner, Xin Qi, Alan Demers, Johannes Gehrke, and Jayavel Shanmugasundaram. 2007. A Unified Platform for Data Driven Web Applications with Automatic Client-Server Partitioning. In *Proceedings of the 16th International Conference on World Wide Web* (Banff, Alberta, Canada) *(WWW '07)*. ACM, New York, NY, USA, 341–350. https://doi.org/10.1145/1242572.1242619

[140] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (San Jose, CA, USA) *(NSDI '12)*. USENIX Association, Berkeley, CA, USA, 14. http://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf Accessed 2020-05-05.

[141] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. 2002. Secure Program Partitioning. *ACM Transactions on Computer Systems* 20, 3 (Aug. 2002), 283–328. https://doi.org/10.1145/566340.566343