The Meerkat Vision: Language Support for Live, Scalable, Reactive Web Apps

João Costa Seco
NOVA LINCS, NOVA School of Science and Technology
Caparica, Portugal
Joao.Seco@fct.unl.pt

Jonathan Aldrich
Carnegie Mellon University
Pittsburgh, USA
jonathan.aldrich@cs.cmu.edu

Abstract

The reactive programming paradigm has become ubiquitous for modern web and mobile app development. But despite its many benefits, today reactive programming is limited to data updates within the client, leaving to the programmer the tedious and error-prone tasks of managing updates to code and synchronizing data updates between reactive clients and a server database. In this paper, we lay out the vision for Meerkat, a tierless, reactive, and live programming language designed to scale to the needs of modern applications. We introduce the language through a chat application which runs on our prototype implementation. We then describe approaches for modularizing and scaling Meerkat programs, customizing tradeoffs between properties such as consistency and availability, supporting local-first software and rich data models, and scaling live updates to full DevOps in software organizations. The Meerkat research program will enable a new era of developing apps that are more responsive, reliable, and evolvable than ever before.

CCS Concepts: • **Software and its engineering → Data flow languages**; **Language types**; **Abstraction, modeling and modularity**; **Software architectures**.

Keywords: Meerkat, Reactive Programming, Live Software Updates, Tierless Programming, Modularity, Scalability, Web, Mobile

ACM Reference Format:
João Costa Seco and Jonathan Aldrich. 2024. The Meerkat Vision: Language Support for Live, Scalable, Reactive Web Apps. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '24), October 23–25, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3689492.3690048

Onward! '24, October 23–25, 2024, Pasadena, CA, USA

ACM ISBN 979-8-4007-1215-9/24/10
https://doi.org/10.1145/3689492.3690048

1 Introduction

Since the earliest days of Graphical User Interface (GUI) application development, developers have recognized the need for a paradigm shift: rather than providing scripted interaction paths in the style of command-line interfaces, GUIs must be responsive to the user. This led to the development of design patterns such as Subject-Observer and Model-View-Controller that facilitate responsiveness. The natural extension of this path is the reactive programming paradigm [12], which specifies how a UI is constructed from persistent application state, and how the app responds to user input by changing that state. When the state changes, the system reacts by updating the UI accordingly. Importantly, this update is done automatically based on what state changed and the user-specified declarative code for deriving the interface from the state—freeing the user from managing callbacks and state consistency as required in the patterns mentioned earlier. Due to these benefits, reactive programming is now the dominant industrial approach to developing apps, with support from major industry frameworks.

Unfortunately the power of reactive programming is limited today because available linguistic support does not scale to the distributed and fast-evolving nature of modern apps. Industrial frameworks such as Angular, React, and Svelte as well as languages like Elm provide a great programming experience when coding up a local app, but real apps require access to cloud data, which typically requires moving outside the reactive paradigm. Industrial reactive database interfaces such as R2DBC [14] exist but require giving up the elegant linguistic support for reactivity provided by systems like Elm and Angular. This lack of support for scale becomes especially painful as programs are evolved, because distributed apps involve multiple interacting programs – clients, servers, and databases – which must all be updated in synchrony (including migration of live data) when the app is enhanced. Modern app frameworks support just one of these genres in isolation, creating massive DevOps challenges for programmers who must carefully plan and execute updates in an unfriendly world that resists synchronized deployment (e.g. via commercial mobile app stores).

In this paper, we lay out the vision for Meerkat, a system we are building that provides linguistic support for seamless reactivity across a distributed system. We start with an overview of the Meerkat architecture (Section 2), followed

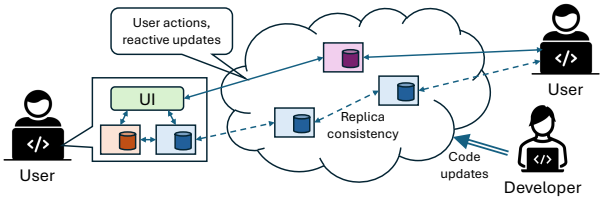

Figure 1. Meerkat architecture vision

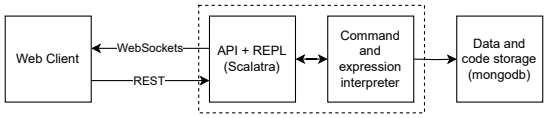

Figure 2. Meerkat current architecture

by a working example that illustrates our vision and highlights an initial implementation of the core ideas (Section 3). Section 4 then lays out our goals and an approach to achieve them: Meerkat builds reactive queries and updates into the language, supporting standard linguistic abstractions while also guaranteeing key safety properties such as termination and glitch freedom. Modular service definitions allow partitioning the backend into reactive microservices, with different distribution, consistency, and privacy policies for different data stores. Live updates to the system provide convenience along with type safety and data consistency guarantees, with support for DevOps essentials like testing and staged deployment. Finally, although Meerkat is designed to provide a great programming experience across app, server, and data development, engineering realities require a design that supports interoperation with off the shelf software technologies, allowing legacy or specialty software components to be incorporated when needed.

Section 5 describes prior work and the limitations that led us to pursue the Meerkat vision. We conclude in Section 6.

2 Meerkat Architecture

Meerkat is a programming model for reactive client-server and distributed applications where a set of resources are made available through simple and reactive APIs implementing user interfaces, data storage, and functionality, while supporting the safe evolution of code and data. It thus provides an abstraction over a complete data-centric system and its evolution process.

Figure 1 depicts our architectural vision for Meerkat. The user interacts with a Meerkat program via their device, which displays a user interface written in Meerkat using web technologies such as HTML and CSS. The UI is generated based on data provided by services, each of which includes internal data storage (blue, purple, and orange boxes). Some services may be local to the client, while others may be hosted in the cloud, and others may be replicated both within the cloud and on user devices. When the user performs an action in the UI, the action is sent to the appropriate service, which updates its internal data store according to the semantics of the action (e.g. adding a row to a data table). The service then sends data updates to all users whose UI was based on the modified data; thus the system ensures that users promptly see updates provided by other users. Any replicated services communicate with each other to maintain consistency according to the developer's desired semantics. Finally, the developer of the Meerkat program can deploy updates to one or more services and clients at any time; these updates are automatically installed into the running system by Meerkat, without ever taking the system down.

At a more concrete level, a Meerkat program consists of a set of named variables holding data and named reactive definitions that compute values based on other variables and definitions. Meerkat data can consist of numbers, text, collections, records, HTML values, and actions that represent monad-like state transformations.

Actions can be executed in response to external events from UI elements or timers, or in respond to a top-level REPL command issued by a Meerkat developer. Meerkat thus defines a two-layer strongly typed language where the top-level language can execute actions and has special commands to create new definitions or update existing ones. Such commands safely evolve the system by updating code and date harmoniously. The bottom-level language is a reactive language that updates the values of top-level definitions whenever their (state) dependencies change.

The Meerkat vision builds on preliminary work, including a proof of concept implementation [10]. This preliminary reference implementation hosts data and code on a centralized application server that provides functionality and a reactive web interface to the user (section 4.3 relaxes the assumption of centralization). The server also provides a special API to typecheck and execute the top-level commands that evolve code and data while maintaining type consistency.

Figure 2 depicts the three-tier architecture of the prototype implementation of Meerkat. This can be viewed as a specialization of the architectural vision depicted in Figure 1; it supports many clients but just one (unnamed) service running on a single server machine. A document-based database (mongodb) stores both code and data, and maintains a persistent representation of the graph of dependencies between variables and definitions. The language interpreter typechecks and interprets the actions and REPL commands, then persists the changes in the database. It also propagates the changes amongst the definitions using the dependency graph and refreshes all subscribers to a given name.

The API+REPL component, which shares the middle tier with the interpreter, exports named variables and definitions as REST endpoints and associates GET and POST requests to

retrieve values and execute commands. The API+REPL component also uses WebSockets to support the user interface reactivity in the case of HTML-like values.

3 Meerkat by Example

To better illustrate the language and the system, we present a simple example of a social network that starts with a simple messaging system and evolves seamlessly to add images to messages. Notice that the system is live (running) at all stages of this example; there is no downtime needed to deploy new functionality. Variables, definitions, and commands are entered by the developer as well-typed blocks and integrated into the system, changing its behavior appropriately. All of the code in this section runs as-is in our implementation; the code in Section 4 is speculative.

Meerkat Example – First Step. The top-level commands in Meerkat are var, table, def, and do. Definitions (def) associate a name to an expression that is reactively evaluated every time one of the names referred in the expression changes (cf. a spreadsheet cell). Stateful stores defined by definitions (var and table) are mutable stores (single values or collections, respectively) that assignments update. Imperative commands (do) trigger assignments and other imperative operations.

The initial state of our example is a messaging system. Its definition starts with a variable to hold a sequence counter and three tables: users, contacts, and messages. The contacts table stores the users of the system with whom one exchanges messages, and the messages table stores the details of the messages sent between users. The following code defines the tables and some initial data:

```
var seq_counter = 0
table users {id:number, name:string, phone:string}
table contacts
  { id:number, owner:number, contact:number}
table messages
  { id:number, fro:number, to:number, text:string}
```

Once these commands are issued in the system, the corresponding storage is created and is live (can be reached from the API). Meerkat's first-class values include action blocks, which wrap delayed imperative constructs like assignments and operations on tables (insert, update, delete). Actions group a set of data updates, and the system executes them atomically and consistently. To allow state updates to depend on reactive data without creating infinite updates, action execution is delayed until it is triggered through the REPL or by a user interaction.

Next, to add new records to tables users and contacts we define functions add_user and add_contact as follows:

Users

id	name	phone
1	Alice	123-4567
2	Bob	234-5678
3	John	345-6789

Contacts

id	owner	contact
1	1	2
2	1	3
3	2	1

Messages

id	fro	to	text
1	1	2	"Hello, Bob!"
2	1	3	"Hello, John!"
3	2	1	"Hi, Alice!"
4	3	1	"Hi, Alice!"

Figure 3. State of the system after the initial setup

```
def add_user id name phone = action {
  insert {id:id,name:name,phone:phone} into users}
def add_contact id owner contact = action {
  insert {id:id,owner:owner,contact:contact} into
    contacts}
```

The insert operation is an imperative operation that adds a new row to the collection (table) specified. Meerkat's proof of concept does not enforce integrity constraints, as that is not its primary focus, but we loosely use numbers as keys, as in the relational model (section 4.6 discusses an extension to referential integrity). The system is reactive; thus, any other program element using the values associated with names that are modified in an action will be updated with the new values computed by these actions. Any client can call the add_contact function to obtain an action value and execute it at the systems' top-level using a special do operation through the REPL, and thus triggering an update to all names connected to the names users and contacts.

```
do add_user 2 "Bob" "234-5678"
do add_user 3 "John" "345-6789"
do add_contact 1 1 2
do add_contact 2 1 3
do add_contact 3 2 1
```

Only Meerkat's REPL or a user interface component can trigger the do operation. Meerkat's type system verifies that circular dependencies between definitions always include an action, whose execution will depend on an external stimulus.

We can similarly define a function send_message that sends a message from one user to another by adding a record to the messages table. Each message receives a new identifier by means of the seq_counter variable that is incremented inside the same action that inserts the message.

```
def send_message fro to text = action {
    seq_counter := seq_counter + 1,
    insert { id: seq_counter,
             fro: fro,
             to: to,
             text: text } into messages
}
```

Function send_message is triggered at the top-level by:

```
do send_message 1 2 "Hello, Bob!"
do send_message 1 3 "Hello, John!"
do send_message 2 1 "Hi, Alice!"
do send_message 3 1 "Hi, Alice!"
```

The resulting state is now populated with users, contacts, and messages, as shown in Figure 3. We can continue to add new tables and variables, but let's consider adding new reactive definitions that inspect the system's state. For instance, the following function retrieves the name of a user given an identifier:

```
def user_name id =
  match get u in users where u.id == id
  with u::rest => u.name
  |    []  => ""
```

Notice above the syntax for queries on tables, inspired by SQL queries, and the use of pattern matching on lists to inspect the results, inspired by sum types in functional languages. The following function retrieves all messages sent to or from a given user.

```
def messages_with owner contact =
  get m in messages
  where m.fro == contact and m.to == owner
     or m.fro == owner and m.to == contact
```

To see the messages exchanged between Alice and Bob, one can define a name to hold the result of the query and then inspect it:

```
def messages_between_Alice_Bob = messages_with 1 2
```

If Alice sends a new message to Bob, the value of the name messages_between_Alice_Bob is automatically updated. If a user interface element refers to that name, the system propagates the change to the client browser displaying that element, and the user sees the updated information.

Meerkat also includes language constructs that mimic HTML to define user interfaces. The following code defines a simple page that lists the messages of a particular user (owner) with one of its contacts (id).

```
def messages_with_page owner id =
 <div>
  <h3>(user_name owner)" and "(user_name id)</h3>
   <ul>
    (map (m in (messages_with owner id))
     <li> (user_name m.fro) ": " (m.text) </li>)
   </ul>
 </div>
```

The actual UI element containing the list of messages between Alice and Bob, given by calling messages_with_page:

```
def msgs_Alice_Bob_page = messages_with_page 1 2
```

A browser accessing the URL associated with the name receives a reactive HTML page containing the corresponding user interface.

The next step in the development of the system is to have a page that lists all contacts with links to the messages exchanged with each of them. The following code defines the page that lists the contacts of a user:

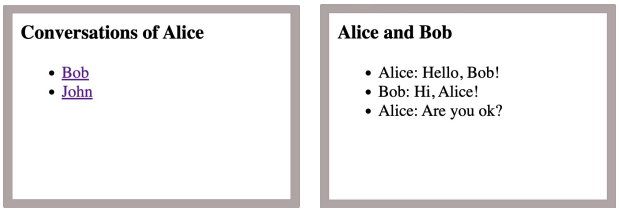

Figure 4. User interfaces for Alice's conversations.

```
def conversations_page id =
 <div>
 <h3> "Conversations of " (user_name id) </h3>
 <ul>
  (map (c in
    (get c in contacts where c.owner == id))
    <li>
      <a href=(linkto messages_with_page id
    c.contact)>
      (user_name (c.contact))
      </a>
    </li>)
 </ul>
 </div>
```

Assuming that the user Alice's user id is 1, we can then define a page that lists Alice's contacts:

```
def conversations_of_Alice = conversations_page 1
```

Our implementation makes this page accessible at a specific URL tied to the name conversations_of_Alice. A URL of the form

http :// servername/app/acQEt/conversations_of_Alice[1]

or, more generally, in the parametrized URL of the name conversations_page, representing a function call, of the form

http :// servername/app/acQEt/conversations_page/1

The (simple) user interfaces produced by these expressions are depicted in Figure 4.

Adding Messages. The system can be further developed by adding new definitions or redefining the existing names to include new features. For instance, the following code redefines the name messages_with_page to include a form that submits new messages:

```
def messages_with_page owner id =
 <div>
  <h3>
    (user_name owner) " and " (user_name id)
  </h3>
  <ul>
   (map (m in (messages_with owner id))
    <li> (user_name m.fro) ": " (m.text) </li>)
  </ul>
  <form>
  <input id="entry"/>
  <button doaction=(send_message owner id #entry)>
     "Send"
  </button>
  </form>
 </div>
```

[1]acQEt is a session identifier corresponding to running instance of the Meerkat program.

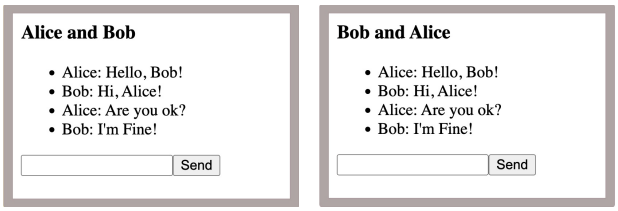

Figure 5. UI for Alice and Bob with a form to send messages and a new message.

Notice in the example above that the language allows the developer to associate an action with a button, using an attribute doaction and an action-typed expression. It is also possible to use the values of input elements in Meerkat expressions. This update in the REPL is immediately propagated, changing the UI of all the active users (Alice and Bob) without any explicit deployment action. The server state remains unchanged. When Alice or Bob clicks the button to send a message, the UI for the multiple users is reactively updated as shown in Figure 5.

Messages with Images - Step 2. The safe evolution of a running system is the core of Meerkat's vision. Top-level commands can add new definitions, as described above, or redefine existing ones. As a trivial extension to our running example, we start by evolving the system to support messages with images, adding a table to store them. The following code extends the system with two new tables

```
table messages_with_images {
  id:number,
  fro:number,
  to:number,
  text:string,
  image:string
}
table images { id:string, url:string }
```

which allow the definition of a new function to insert images:

```
def insert_image id url = action {
  insert { id:id, url:url } into images
}

do insert_image "1" ""
do insert_image "2" "/images/cat1.jpg"
do insert_image "3" "/images/cat2.jpg"
do insert_image "4" "/images/dog1.jpg"
do insert_image "5" "/images/dog2.jpg"
```

We create the first record with an empty string, the default value, to represent the absence of an image. To start using messages with images, we need to redefine the send_message function to include an empty image, and we don't need to change the user interface.

```
  def send_message fro to text = action {
    seq_counter := seq_counter + 1,
    insert { id:seq_counter,
             fro:fro, to:to,
             text:text, image:"1"} into
     messages_with_images
  }
```

In most situations, it is possible to evolve a system step by step without breaking the system, i.e., without inconsistent intermediate states. First, if no types are changed, the system remains (type)consistent. Suppose types are changed, and the replacement of a definition is at a root in a dependency graph. In that case, it is possible without disturbing the system because no other definition uses it. However, sometimes breaking changes are inevitable, e.g., when changing the types of names in mutually recursive definitions. In other situations, the result may not be ideal. For example, when send_message is updated as described above, new messages will not be shown in the user interface because the UI gets information from the old messages table. To support mutual recursion and to make updates seamless for users, it is possible to group several commands in an atomic block. By doing so, all commands are executed in isolation, the inconsistent states happen but are not observed, and the system is consistent at the end of the block.

```
atomic {
  def insert_message id fro to text image =
    action {
     insert { id: id,
              fro: fro, to: to,
              text: text, image:image }
    into messages_with_images
  }

  do map (m in messages)
    ( insert_message m.id m.fro m.to m.text "1" )

  def send_message fro to text = ...

  def messages_with owner contact =
    get m in messages_with_images
    where m.fro == contact and m.to == owner or
    m.fro == owner and m.to == contact
}
```

Notice that the do command iterates over all existing messages and produces an action for each one. It then iterates over all actions and executes them. This copies all messages from the previous table to the new one. By additionally defining the function that reads the messages and the function that sends an image without a picture, we can ensure that the system is always in a consistent state with respect to both code and data.

Atomic updates are convenient for the programmer, but in the presence of large datasets, they may result in noticeable pauses as ordinary operations are halted to guarantee global synchronization. For some applications, this is acceptable; a lot of modern mobile and web apps do go down for short maintenance breaks. For applications that must be continuously available, though, this is a problem, one that we return to in section 4.7.

Once the above is complete, we can equip the system to send a message with an image by defining a new function, assuming an image already exists in the images table.

```
def send_message_with_image fro to text image =
    action {
      seq_counter := seq_counter + 1,
      insert { id:seq_counter,
               fro:fro, to:to,
               text:text, image:image } into
    messages_with_images }
```

Finally, to make the system compatible with older or unchanged elements, we can redefine the name messages from a table to a kind of a "view" over the new table that ignores all the images in the messages.

```
def messages = map (m in messages_with_images)
    { id:m.id, fro:m.fro, to:m.to, text:m.text }
```

The only step that is still needed to upgrade the system is to redefine the user interface. At this point, the UI is the same, but emits messages to the new table, with the default value for the image field. First, we define a function that produces the code to display the image and ignores the empty image. Then, we update the page to receive an image URL and call the appropriate function to display the image, as shown in Figure 6.

```
def image_of image =
  if image == "1" then <span></span> else
    (match get i in images where i.id == image
      with i::rest => (<p><img src=(i.url)/></p>)
      | [] => (<span></span>))

def messages_with_page owner id =
  <div>
    <h3>
    (user_name owner) " and " (user_name id)
    </h3>
    <ul>
      (map (m in (messages_with owner id))
        <li>
        (image_of m.image)
        <p>(user_name m.fro) ": " (m.text)</p>
        </li>)
    </ul>
    <form>
      <div> "Text:" <input id="entry"/></div>
      <div> "Image:" <input id="image"/></div>
      <button doaction=
          (send_message_with_image owner id #entry
    #image)>
          "Send"
      </button>
    </form>
  </div>
```

At this point, the interface supports both messages and images. The system was evolved seamlessly from the original version without ever being taken down or losing any data.

Notice that the current implementation is a proof of concept of the system we envision. Currently, it supports some convenient extensions like CSS and JavaScript code that can be associated with the HTML values. More broadly, its design is compatible with the basic definitions written in other mainstream (typed) languages. Ideally, Meerkat would work as a component-based platform where components can be reactively interlinked, provided they implement and propagate the reactive behavior. One strong candidate for a base language for components would be the Kotlin programming language, which compiles to multiple contexts.

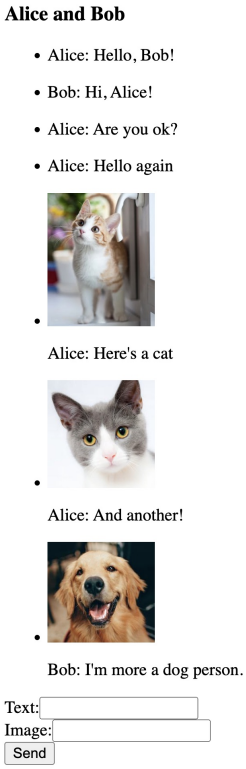

Figure 6. New user interface for Alice's conversation with Bob with images.

The prototype system supports reactive execution of programs and live updates, but it is not yet practical for building applications at scale due to a number of limitations in the language and system architecture. In particular, the choice of hosting code and data in the same database is a simplification that allows us to focus on the core language and system design. The problem of code and data migrations has been studied for a long time and has a number of solutions. The Meerkat vision is to provide a language that supports the safe evolution of code and data in conjunction with existing technological solutions for code and data migration. Our goal is to have high-level declarative construction operations that support seamless construction and evolution of applications and are supported by efficient operations in underlying systems or languages. In the following section, we describe a vision that scales up reactive, live programming in Meerkat and outline a research program for overcoming the limitations of the current implementation.

4 Meerkat Vision

In this section, we describe Meerkat's vision, building on the design sketched above to support live programming at enterprise scale.

4.1 Linguistic Support for Reactivity

A core principle of Meerkat, illustrated in the example above, is that support for reactivity is *linguistic*. It is possible to provide reactive programming through library abstractions, for example, as provided by React[2]. However, the core abstraction of reactive programming is reacting to changing data and providing reactivity as a linguistic construct supports that model transparently. This transparency supports clear, declarative code that directly captures programmers' intent.

The Meerkat vision differs from prior research on reactive functional programming, as supported by Elm [9] and Flapjax [20], in that our updates to data are imperative. The enthusiastic adoption of industrial frameworks like React and Angular suggests that a mutable data model is an abstraction that matches programmers' mental models well. Meerkat's approach is closer to what is present in recent versions of Redux Toolkit[3], a state management framework that changed from a purely functional style to a combination of imperative and reactive programming constructs. While abstractions such as event streams in Flapjax or messages in Elm can function similarly, they create an impedance mismatch with the way that programmers think about data, and result in awkward boilerplate code. For example, the Counter.elm example from Elm's website[4] has button clicks generate Increment and Decrement messages, which are then passed to an update function along with the previous model in order to compute the new model:

```
type alias Model = Int

initialModel : Model
initialModel = 0

update : Msg -> Model -> Model
update msg model =
    case msg of
        Increment ->
            model + 1

        Decrement ->
            model - 1
```

the code in the *View* part:

```
button [ onClick Decrement ] [ text "-" ]
```

While there are advantages to functional programming, in our view the code obscures what is really going on: the programmer's mental model is of incrementing and decrementing a counter. Compare the equivalent code in Meerkat:

```
var counter = 0

def increment = action { counter := counter + 1 }
def decrement = action { counter := counter - 1 }
```

the code in the *View* part:

```
<button doaction = (decrement) > "-" </button>
```

Furthermore, Meerkat aims to capture the server-side data model, where imperative updates are standard. Our imperative data model can express client and server data equally well, while functional reactive programming requires programmers to mediate between the two with commands or events that result in calls to functions with communication as a side effect. For example, Elm's tutorial on getting data from a server via JavaScript does this with an update function that generates side-effecting commands along with a new model:

```
update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
  case msg of
    SendDataToJS ->
      ( model, sendData "Hello JavaScript!" )
    ReceivedDataFromJS data -> ( data, Cmd.none )
```

Meerkat's language layer abstracts data location and requires none of this boilerplate–the same Meerkat counter code given above works regardless of whether the counter is on the same machine or a different server. The actual location and availability can be guided by a different set of requirements like data sharing, computational needs and capabilities, data confidentiality, communication, etc. We do adopt ideas from functional reactive programming to represent computation of output from changing data; it is clear that, in this setting, functional programming is the most direct abstraction and provides a clean, safe semantics for user interface code.

Besides supporting a cleaner programming model with a close correspondence to the mental models of programmers, linguistic support for reactivity allows Meerkat to provide programmers with guardrails to get their reactive programs right. Our design uses def to represent reactive definitions, distinguishing them from both data definitions (which use var or table) and imperative updates with action. Our type system allows flexible composition, so that actions can use reactive definitions to compute data definitions, and reactive definitions can define deferred actions. However, reactive definitions cannot *trigger* actions, avoiding self-triggering loops. Furthermore, our type system ensures that reactive definitions are not circular [10].

We believe the developer user experience (UX) of reactive programming should provide all the powerful abstractions provided by modern programming systems. Therefore, as part of supporting linguistic reactivity, a priority for Meerkat is to support rich, type-safe programming abstractions, including higher-order functions in reactive definitions and actions, rich data structures, and modular decomposition of definitions, data structures, and actions. While higher-order

[2] https://react.dev/
[3] https://redux-toolkit.js.org/
[4] https://elmprogramming.com/

functions are already supported in our prototype, the other features pose interesting design and research challenges, as discussed below.

Functional reactive programming is thriving in the context of JavaScript and web programming. Therefore, we envision a mechanism that allows JavaScript web components to be integrated into Meerkat applications, with clear interfaces that preserve the reactive semantics across client and server borders and services. In the current prototype, it is possible to add JavaScript code and CSS elements to applications, but we envision an integration that for example allows programmers to seamlessly import TypeScript React components and integrate them with a line or two of typechecked Meerkat code. Supporting integration with TypeScript code will also provide a way to support functionality that is not part of the Meerkat execution model. For example, our model does not support threads or timers that could be used to do things like generate periodic events that update application state, without the user taking any action. A component written in TypeScript, or perhaps provided as a server-side plugin in a language like Rust, could generate such events and trigger Meerkat actions in the same way that a user-facing interface, or the system's REPL would.

4.2 Reactivity Across the Software Stack

A number of research languages and industrial frameworks provide seamless reactivity on the client, but many real applications store data on servers or in the cloud. This typically forces programmers to depart from the reactive paradigm to explicitly request data and updates from the server, creating an impedance mismatch that results in repetitive, boilerplate code, which ends up being error-prone as well since programmers are tasked with the difficult problem of keeping distributed data consistent between the client and server.

Industry and researchers have recognized this problem and have begun to address the issue, but their solutions are limited by not fully embracing reactivity via linguistic abstractions. For example, the Meteor framework [24] connects reactive front-end components (e.g., written in Angular or React) to a MongoDB database, via reactive abstractions in JavaScript. The Reactive Relational Database Connectivity (R2DBC) [14] specification provides a reactive programming API to relational databases, allowing developers to access their data with reactive SQL queries. These projects offer concrete benefits to developers by extending reactive programming to server side programming, and demonstrate that the industry is interested in reactive programming that spans beyond the client. However, they still provide an impedance mismatch for programmers who are using linguistic reactivity on the client. For example, a programmer using Angular's compiler support for reactive programming will find that accessing Meteor data requires observables to mediate between data updates from Meteor and Angular's natively reactive TypeScript fields. Here's code from the Meteor/Angular tutorial that listens to MongoDB-compatible `Collection`s via Meteor and turns them into rxjs `Observable`s that update the UI whenever chats or their messages change:

```
import { MongoObservable } from 'meteor-rxjs'

const Chats =
  new MongoObservable.Collection<Chat>('chats');
const Messages =
  new MongoObservable
  .Collection<Message>('messages');
  // in the implementation of ChatsPage:
  ngOnInit() {
    this.chats = Chats
      .find({})
      .mergeMap((chats: Chat[]) =>
        Observable.combineLatest(
          ...chats.map((chat: Chat) =>
            Messages
              .find({chatId: chat._id})
              .startWith(null)
              .map(messages => {
                if (messages)
                  chat.lastMessage = messages[0];
                return chat;
              })
          )
        )
      ).zone();
  }
```

It's not necessary to understand the details of this code; the point is that there are layers of abstraction required to adapt from Meteor's cross-platform reactive abstractions and the ones that Angular uses, all of which creates boilerplate for programmers to write. The mismatch is less pronounced when using React, which has a more natural integration with Meteor's reactive abstractions using hooks [25]. However, the code is still more complex than it would be in Meerkat, where reactivity is built-in. Furthermore, React is already less 'linguistic' in its support for reactivity than either Angular or Meerkat; state is represented with getter/setter pairs, for example, rather than fields with assignment and update. For systems using R2DBC, there are similar issues: it provides reactive SQL queries, but they are written in SQL instead of the reactive front-end language, so adaptation is necessary.

The lack of linguistic support – and particularly typechecking – in existing full-stack reactive frameworks prevents them from catching the kinds of errors that our type system can identify. For example, if the observable code that ties Angular to Meteor accidentally invokes a function that updates the database, a nonterminating reactive loop may result. It is also a barrier to providing programmers effective support for the live evolution of reactive programming systems, since there is no system that can analyze and evolve code on both the client and server.

By providing reactive linguistic abstractions that span machine boundaries, Meerkat is able to provide a convenient and consistent programming interface for server data structures and client UI code alike. Furthermore, Meerkat's type

system is able to detect static errors in both the construction and the evolution of reactive applications.

4.3 Services: Scaling Up Reactive Data

A primary limitation of our current prototype is that all the data is declared in a single module and hosted on one server. This made implementation of a seamless reactive programming language easier, but it limits scalability in multiple dimensions: size of data, code complexity, request throughput, and reliability. Meerkat will overcome these limitations by introducing the idea of services.

A *service* encapsulates a set of data declarations and associated reactive definitions and actions that are managed together. It is the reactive, linguistic analog of a microservice [16]. Data inside a service is declared using the same var and table declarations described earlier. These can be exposed to clients, but more typically, a service hides them and instead provides a high level *service signature* made up of reactive definitions and actions. For example, a message service might have the following signature:

```
service sig message_service {
  type Message =
    {id:number, fro:number, to:number, text:string}
  action send_message(to:number, fro:number,
    text:string)
  def messages_with(owner:number, contact:number)
    : Message list
}
```

These are of course just the same functions from our running example, now given explicit types and nested in a service definition—and they can be defined exactly as before. The messages_with definition is reactive: clients that use its definition are notified whenever its value changes. We treat send_message slightly differently: instead of a definition that returns an action, we treat it as a parameterized action that returns no result but may have a side effect on the service. The reason is that the client will not invoke the message service until it is ready to send a message—we do not want to do a round trip to get the action and then another one to invoke it. After an action modifies data within the service, changes to reactive definitions will be propagated to clients as appropriate.

From a software engineering point of view, a service acts as a module or object: it has an abstract interface, permitting its data representation to be changed without affecting client code. Separate evolution of different parts of an application's service is a critical part of the motivation for microservices, and will provide similar benefits for reactive programming.

Services can be composed, allowing one service's definitions and actions to be defined in terms of other services. This echoes the composition of microservices in today's cloud computing systems. To enable static checking, the signature of a service must capture the dependencies of a service's definitions. When a composed system is built, a check that is global to the program but uses only signatures, not implementations, verifies that there are no dependency cycles in reactive definitions.

4.4 Located Services and Thick Clients

Our service construct is motivated by microservices, and in our design, services can be instantiated independently; each service is the equivalent of a main program in C. Instantiating a service also instantiates all the other services it depends on; we propose an import construct to make dependencies explicit. In the following code, instantiating the messages service also creates an instance of the accounts service (code not shown).

```
service messages: message_service {
  import accounts // messages depends on accounts
  ...
}
```

To allow services that interact to be instantiated separately, services can take other services as parameters instead, much like a functor [18]:

```
service messages(accounts: account_service):
    message_service {
  ...
}
```

At deployment time, we envision that services will be identified by URLs. Thus when instantiating the messages service, we pass a URL for an account_service that is already instantiated.

Local-First Software. An important application of services is to support Local-First Software [15], a design philosophy that emphasizes users' choice in how their private data is managed–in particular the ability to restrict that data to their own devices. Meerkat's design will support declaring a service for each user that encapsulates their data, and is stored on their devices. Since one application may have an unbounded number of users, this requires a service to be dynamically instantiated whenever a new user joins the system. To this end, the client application can be represented as a single service that includes data members as well as a user interface and takes any external services it uses as parameters. A local-first messages client might look something like the following code:

```
service message_client(dir:directory_service) {
  table my_messages { ... }
  action send_message to text = ...
  def sent_messages contact = ...
  def recd_messages contact = ...
  def $ = <div> ... </div>
}
```

Here, the user interface is represented as HTML, as described above, using a distinguished name $ to represent the application home page. The message_client can be deployed as a standard native application (with web components under the hood) or as a web application. In the latter case, the server might look something like this:

```
service message_server {
  import directory:directory_service

  def $ = message_client(directory)
}
```

Now if the user loads the root URL of the message server (/, represented again as $), the code for the service, compiled as a package of HTML and JavaScript, is downloaded to the client browser and shown to the user. The table in the message_client service is stored in local storage in the client-side browser.

Instead of being parameterized by services, local clients could also connect and disconnect to them dynamically. A language primitive for creating a reference to a service from its URL would enable this functionality. Simply disconnecting and reconnecting to services allows local-first software to run in offline mode. A more sophisticated approach is to use replicated services that use CRDTs to tolerate disconnected operation, as described in the next subsection.

4.5 Customizable Semantics for Services

The abstraction of services also supports different tradeoffs between consistency, reliability, scalability, and other quality attributes. Some services, such as user account management and authentication, must provide a high degree of consistency but are not expected to deal with a high request rate. Other services must be highly scalable and can sacrifice some consistency to achieve that goal.

Plug-in Service Implementations. We propose to support this diversity of semantics by providing a plug-in extension mechanism for services. The plug-in interface allows services to be implemented in multiple ways. One implementation of a data storage service might use a conventional database, leveraging the R2DBC reactive database interface [14], with the accompanying reliability and consistency properties. This could be denoted within Meerkat as follows:

```
extern("RDBC") service msg_database {
  action send_message to text = "INSERT INTO ..."
  def sent_messages contact = "SELECT ..."
  ...
}
```

The declaration above indicates that Meerkat should search for and loads a service implementation named RDBC. Instead of providing an implementation of the definitions and actions in Meerkat, a string is provided that is interpreted by the plugin. In the case of RDBC, definitions exported by the service map to programmer-defined reactive queries over the database, whereas actions might map to SQL INSERT statements or to stored procedures. There are limitations to this approach—namely the restriction to databases with reactive query support and the restriction of service definitions to operations the database can support—but it allows developers to use off-the-shelf database technology if that best meets their needs.

Native Meerkat Service Implementations. Using plugins to connect to conventional databases is valuable as it allows developers to use existing commercial infrastructure. However, synchronizing code updates with an external tool such as a database may be hard (though perhaps not impossible–it's one direction we may investigate in future work). We therefore plan to provide distributed reactive database implementations with desirable semantics and scalability properties for our setting. A first step in this direction is Historiographer [13], a novel distributed reactive database algorithm that provides causal consistency—a relatively strong consistency model that nevertheless avoids the global synchronization that is inherent to full serializability. In Historiographer, only the names used in a transaction are read- or write-locked, allowing fully parallel queries and parallel updates to orthogonal data. Deadlocks are handled with wait-die deadlock avoidance. Historiographer keeps a history of recent updates to each definition in order to provide causal consistency and glitch freedom. Experiments validate what one would expect: on microbenchmarks, Historiographer scales better than state of the art algorithms that provide serializability.

We are currently extending Historiographer to support code updates. Code updates are typechecked when they are written to ensure that the overall system after updates is well-formed. In our proposed approach, code updates compete with other transactions, acquiring exclusive locks on definitions that are updated and read locks on definitions that an update depends on or that depends on it. As actions can be triggered simultaneously with updates, there is a race to acquire the appropriate locks. If an action loses that race, it may no longer typecheck against the new definitions. Therefore, actions track versions of definitions, which are incremented on code updates. We check definition versions to detect the possibility of incompatibility and re-typecheck. Actions that still typecheck can be run; other actions are canceled, with a notification to the user (an exception-like mechanism to be designed).

Similar problems can happen when conflicting code updates race, and can be solved in a similar way: if version numbers show that relevant code has changed since a code update was last typechecked, it must be typechecked again before being applied. In many cases code updates require more coordination than actions, meaning they will be more expensive to apply. However, we assume they will also be much rarer–a popular commercial application might see code updates once per day if it is developed in a highly agile fashion, but its users will be performing millions or billions of operations per day.

Weakening Consistency. Some applications do not require strong consistency and glitch freedom, or require it for only parts of their applications. In this case, developers

may prefer eventual consistency if it provides better scalability, and they may prefer updates to become immediately visible despite the possibility of glitches to provide the user with the most up-to-date information. We plan to provide annotations that relax the consistency guarantees provided by Historiographer, and modifications to the algorithm that support this.

For example, consider a text editing application that provides spell-checking functionality from the server. In the classic definition of glitch freedom, the results of inserting a character would not be shown to the user until all reactive code triggered of that insertion—including server-side spell checking—is complete. Historiographer implements glitch freedom by holding back partial updates from being displayed until all updates from the same transaction are available. Of course this is unacceptable in the document editing setting, as users wish to see the effect of their edits immediately, and spell checking can come later. Our approach would annotate the definition of the displayed document text as `@Eager`, meaning that reactive updates affecting this definition are never held back by Historiographer. Users may add a character to the document and see the document, then learn shortly afterwards that as a result a word is mis-spelled. This is technically a glitch, but it is acceptable in this context. Our type system will verify that glitch-free definitions do not rely on definitions or service implementations that cannot provide those semantics.

If the text editor is to be collaborative, we may want to implement it with conflict-free replicated datatypes (CRDTs) [23] so that it is maximally responsive. A simple example of a CRDT is an insert-only table. We plan to provide an `@InsertOnly` annotation that restricts the program from updating or deleting rows in the table, supports an eventual consistency semantics for the table, and provides a corresponding efficient run time implementation. Eventual consistency supports (temporarily) offline applications well, an important use case for local-first software.

In addition to the two annotations sketched above, we plan to support a broader set of annotations that support commonly desired consistency/performance tradeoffs in real applications. This will require careful thinking (and consideration of prior work [27]) to determine what semantics programmers can rely on in programs that have both causally-consistent and eventually-consistent data structures.

Implementation Approach. Our current implementation of Historiographer is an interpreter, which sacrifices some performance but not too much given that distributed communication is the primary bottleneck. We plan to explore a compiler targeting Hydroflow [22] to gain improved performance. It will be a just-in-time compiler in order to support recompilation when code updates occur.

Building on prior work [2], Meerkat will provide a configuration language for services that links each service in the program to the developer's selected implementation. The configuration will have an extensible section for each service that allows additional provider-specific deployment information to be specified; this might, for example, specify the replication strategy in a future, replication-aware version of Historiographer, or for an existing cloud database provider. One strategy might be `shard by messages.to, replicas 3`, instructing the database to divide up (shard) the message data by the `to` field so that it can be accessed in parallel, and replicate each entry on 3 servers for reliability purposes. The configuration language will also associate each service provider with a well-known consistency semantics, so that compositions can be checked. For example, a service provider that guarantees causal consistency should not rely on a separate service whose provider guarantees only eventual consistency, because then the first provider will not be able to honor its own consistency guarantee.

4.6 Rich Data Models

While the current prototype of Meerkat does not provide properties such as referential integrity, many applications rely on this property, and we would like to make it an option in our system. While conventional reactive databases can provide this, we hope to provide a more scalable version by starting with Historiographer [13], a recently-developed distributed reactive update algorithm that provides causal consistency and glitch freedom while avoiding any need for global coordination. In our proposed approach, the developer can specify whether referential integrity is important. If integrity is not specified, then queries might fail to return results if a key is missing.

If integrity is specified, however, users can annotate certain columns as primary keys (using `@primary_key` on the field), and others as foreign keys referring to a primary key in another table (e.g. `@foreign_key(user_service.users.id)`). We will keep a count of foreign key uses of each primary key in the system, and ensure the primary key cannot be removed from the table unless the count is zero. The causally consistent transaction update mechanism in Historiographer will be used to ensure the count is consistent with the actual number of foreign key uses. If the user tries to do so in an action, the action will fail. Our compiler will analyze actions for the possibility of failure—i.e. whether the action can remove or change a primary key—and require the user to provide error-handling code in that case.

4.7 Distributed, Live Updates, and DevOps Support

By modeling an entire end-to-end system based on declaratively built reactive dataflows, Meerkat can isolate the key axes on which flexible and incremental updates hinge. Our `do` and `atomic` blocks support discrete transitions between consistent states of the system in order to enable incremental construction and evolution of a system. Here we benefit from

the well-known advantages of functional programming languages in our declarative definitions, while still supporting a notion of mutable state that is core in data-centric systems.

A key challenge in scaling safe, live updates from single-server systems to more arbitrary distributed architectures is coordinating updates across the many servers that may host individual services. Some of our diverse approaches to data consistency support substantial independence between services. Likewise, we anticipate that many code and data evolution steps, such as adding a new definition, cannot break existing code and therefore can be applied one service and one server at a time. Here, the structure of updates in Meerkat provides an opportunity to cleanly intersperse the regular execution of system operations with code and data updates. Therefore, the log of all construction operations uniquely defines the current state of evolution of the system. In principle, it should be possible to propagate operations and replay this log in different environments in the style of operation-based CRDTs [23]. We envision this mechanism as a foundation for language-based DevOps operations, safely staging the different development environments (development, quality assurance, and production).

We also recognize that atomic updates may be nonmonotonic, and these will require more global synchronization—at least within a service, but in cases where the service's interface changes, perhaps beyond it as well.

Meerkat will include tools for analyzing updates to see whether they can be deployed incrementally, as well as a coordination layer that operates above the level of actions to permit globally consistent, non-monotonic updates. In the case of global updates, it is likely that users of a system may see a brief pause or interruption as the update is deployed. Our hope, however, is that the update automation provided by Meerkat can reduce these pauses dramatically compared to the way system updates operate today. Furthermore, the typechecking of updates eliminates large classes of errors that can occur during more manually-managed software updates in the state of the practice today.

For applications that manage large datasets and must be continuously available, programmers can mimic the strategies that are currently used to update online applications: creating new tables but continuing operations against the old ones while data is being copied over. Once the data is present in both old and new data structures, code can be switched to use the new data structures with minimal pauses.

Such a strategy is common practice in companies that deal with large amounts of data, for example as documented by Jacqueline Xu at Stripe [26]. The example presented in section 3 is a simplified version of this strategy; the main missing element is a dual write strategy that allows the old database table to be used in parallel with the new one, and a separate process to copy the data.

Coding this up in Meerkat is more work for developers than using atomic updates, but it is much easier than the status quo, where developers do not benefit from the many facilities provided by Meerkat: a tierless view of the system, atomic updates that are still used for smaller steps, the guarantee that updates are well-typed, and the DevOps facilities proposed above. In addition, we anticipate that many data-centric update operations can be automated and synthesized from declarative specifications based on prior research in this area [1].

More broadly, a language-based model that supports live updates can be used to design new DevOps strategies, for instance, to automatically update versions in clients that are temporarily off-line or to program a rollout strategy that "pre-stages" new versions to selected sets of clients, using mock versions of the core data elements.

5 Related Work

Meerkat's core vision is to support live programming at scale by abstracting the dataflow into a base structure that supports rigorous reasoning about type safety and non-recursive definitions. Other works have approached parts of this problem from different angles.

We take inspiration from languages implementing tierless programming, like LINKS [7] and Ur/Web [5], which provide code for three tiers based on a single code source. Like Meerkat, Ur/Web also adheres to the functional reactive paradigm. In Meerkat, the location of data and computation is further abstracted than in these languages. Nevertheless, Meerkat's focus is to pave the ground for the safe evolution of code and data, which is not supported by these languages. The refactoring of code and its more dynamic handling at the language level is critical in other languages like Unison [6], which is a distributed, pure functional language. Unison focuses more on the distribution of code and data, ensuring that all definitions use the correct version of a function through a hashing method despite their names changing. Meerkat takes a different approach and safeguards the redefinition of code by analyzing the dependency graph and disallowing illegal updates. All updates are still possible but must follow a particular protocol.

Other languages and systems explicitly adopt the tier division between client and server, like Riffle [17], and define synchronization transformations between the two. Meerkat's approach is to completely abstract the location of data and computation, allowing the developer to focus on the application's logic. The compiler determines the location of data and computation in an orthogonal process. We present a generic definition of data services that can be configured to capture more situations than the tier division. On the pragmatic side, Riffle can be used directly with mainstream reactive technologies.

From a DevOps perspective, the work on the sound evolution and adaptation of microservices in the setting of low-code platforms [8] shows that most modifications can be captured and automatically adapted. The same happens in Meerkat, where the type system controls which changes are legal and which will break the system's operation. The type system helps identifying which elements can be changed individually and which need to be grouped in a bundle, to make each evolution step safe.

To support the existence of different versions of the same functionality in Meerkat, we need to define transformations between different representations. This structured approach to the evolution of software is also present in works like Carvalho and Costa Seco's [4] and Campinhos et al. [3]. The difference is that Meerkat's transformations are always live and are not inlined by the compilation process to attain a single version of the system.

In a sense, Meerkat defines the high-level structure of a cloud application. Language-based approaches to decentralized computing have been presented, for instance, in CPL [2], which provides a configuration language for cloud computing, and in Margara and Salvaneschi [19], who focus on a distributed reactive mechanism with glitch freedom guarantees.

Reactive programming languages are at the base of systems like Meerkat. Languages like Elm [9] provide a reactive programming model for the client side but do not extend it to the server side. Flapjax [20] allows the creation of dynamic connections to data that cross to the server. Meteor [24] is a full-stack reactive framework that provides a reactive model for the client and the server but does not provide a language model. R2DBC [14] provides reactive programming APIs to relational databases but does not provide a unified language model across the client and server. Meerkat provides a typesafe and unified reactive programming language model across services and tier borders. A language model allows for the creation of new abstractions, like the synchronized update of code and data, a notion of consistency between actions and persistent data schemas. However, Meerkat's primary goal in using a reactive model is to provide ground for the safe evolution of code and data.

An alternative to providing built-in language support for reactive programming is to build an embedded DSL in languages that provide rich type system and metaprogramming facilities. The ReScala project embeds reactive constructs in Scala as a library that approaches the developer UX of a reactive programming language [21]. Notably ReScala supports distributed reactive programming, including the development of a novel distributed update algorithm [11] that was the point of departure for Historiographer [13]. The ReScala approach has the benefit of being easily adoptable as it lives in the ecosystem of Scala, a language with many available libraries and a knowledgable user community. However, because it does not build in reactive semantics, it lacks some of the type checking benefits of Meerkat and does not support type safe live updates.

6 Discussion and Conclusion

This paper presented the Meerkat vision: a language and programming system for large-scale applications that leverages reactive programming to provide a great programming experience for people building and updating distributed applications. We illustrated Meerkat via an example of a web application which can be seamlessly and safely evolved live even after its initial construction and deployment.

Meerkat's abstractions are possible due to the declarative nature of the functional reactive style adopted. The language model captures the data flows that characterize any data-centric application from a bird's eye perspective. Such a model allows reasoning about the changes needed to evolve a system.

The vision presented in this paper is to take the current prototype to a level where the native linguistic support for reactivity can be used to reason about live updates, DevOps, data placement, and richer data management models. We hope that when this vision is realized, Meerkat will be immediately useful to distributed application developers, and in addition that the research program described here will lay the foundation for the next family of industrial reactive programming languages as well as future academic research in live programming and distributed and reactive systems.

Acknowledgments

This work is partially supported by EU Horizon Europe under Grant, Agreement no. 101093006 (TaRDIS), NOVA LINCS UIDB/04516/2020 (10.54499/UIDB/04516/2020) and UIDP/04516/2020 (10.54499/UIDP/04516/2020) with support of FCT.IP, the US National Science Foundation under grant no. CCF1901033, and the US Department of Defense. We thank Selva Samuel and the anonymous reviewers for feedback that substantially improved the paper.

References

[1] Sara Almeida. 2021. *Type-Driven Synthesis Of Evolving Data Models*. Master's thesis. NOVA School of Science and Technology. http://hdl.handle.net/10362/135849

[2] Oliver Bračevac, Sebastian Erdweg, Guido Salvaneschi, and Mira Mezini. 2016. CPL: a core language for cloud computing. In *Proceedings of the 15th International Conference on Modularity* (Málaga, Spain) *(MODULARITY 2016)*. Association for Computing Machinery, New York, NY, USA, 94–105. https://doi.org/10.1145/2889443.2889452

[3] João Campinhos, João Costa Seco, and Jácome Cunha. 2017. Type-Safe Evolution of Web Services. In *2017 IEEE/ACM 2nd International Workshop on Variability and Complexity in Software Design (VACE)*. 20–26. https://doi.org/10.1109/VACE.2017.6

[4] Luís Carvalho and João Costa Seco. 2021. Deep Semantic Versioning for Evolution and Variability. In *Proceedings of the 23rd International Symposium on Principles and Practice of Declarative Programming* (Tallinn, Estonia) *(PPDP '21)*. Association for Computing Machinery, New York, NY, USA, Article 21, 13 pages. https://doi.org/10.1145/3479394.3479416

[5] 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)*. Association for Computing Machinery, New York, NY, USA, 153–165. https://doi.org/10.1145/2676726.2677004

[6] Unison Computing. 2024. *Unison*. https://www.unison-lang.org/ Online, accessed on 2024-07-18.

[7] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2007. Links: Web Programming Without Tiers. In *Formal Methods for Components and Objects*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 266–296.

[8] João Costa Seco, Paulo Ferreira, Hugo Lourenço, Carla Ferreira, and Lucio Ferrão. 2020. Robust Contract Evolution in a TypeSafe MicroServices Architecture. *The Art, Science, and Engineering of Programming* 4, 3 (Feb. 2020). https://doi.org/10.22152/programming-journal.org/2020/4/10

[9] Evan Czaplicki and Stephen Chong. 2013. Asynchronous functional reactive programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 411–422. https://doi.org/10.1145/2491956.2462161

[10] Miguel Domingues and João Costa Seco. 2015. Type safe evolution of live systems. In *Workshop on Reactive and Event-based Languages & Systems (REBLS'15)*. http://ctp.di.fct.unl.pt/~jcs

[11] Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. 2014. Distributed REScala: an update algorithm for distributed reactive programming. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 361–376. https://doi.org/10.1145/2660193.2660240

[12] Conal Elliott and Paul Hudak. 1997. Functional reactive animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming* (Amsterdam, The Netherlands) *(ICFP '97)*. Association for Computing Machinery, New York, NY, USA, 263–273. https://doi.org/10.1145/258948.258973

[13] Julia Freeman and Timothy Zhou. 2023. Historiographer: Strongly-Consistent Distributed Reactive Programming with Minimal Locking. In *Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity* (Cascais, Portugal) *(SPLASH 2023)*. Association for Computing Machinery, New York, NY, USA, 31–33. https://doi.org/10.1145/3618305.3623597

[14] Ben Hale, Mark Paluch, Greg Turnquist, Jay Bryant, and Elena Felder. 2022. *R2DBC - Reactive Relational Database Connectivity*. https://r2dbc.io/spec/1.0.0.RELEASE/spec/html/ Online, accessed on 2024-04-21.

[15] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. 2019. Local-first software: you own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Athens, Greece) *(Onward! 2019)*. Association for Computing Machinery, New York, NY, USA, 154–178. https://doi.org/10.1145/3359591.3359737

[16] James Lewis and Martin Fowler. 2017. *Microservices: a definition of this new architectural term*. https://martinfowler.com/articles/microservices.html Online, accessed on 2024-04-25.

[17] Geoffrey Litt, Nicholas Schiefer, Johannes Schickling, and Daniel Jackson. 2023. Riffle: Reactive Relational State for Local-First Applications. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (San Francisco, CA, USA) *(UIST '23)*. Association for Computing Machinery, New York, NY, USA, Article 76, 16 pages. https://doi.org/10.1145/3586183.3606801

[18] David MacQueen. 1984. Modules for standard ML. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming* (Austin, Texas, USA) *(LFP '84)*. Association for Computing Machinery, New York, NY, USA, 198–207. https://doi.org/10.1145/800055.802036

[19] Alessandro Margara and Guido Salvaneschi. 2014. We have a DREAM: distributed reactive programming with consistency guarantees. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems* (Mumbai, India) *(DEBS '14)*. Association for Computing Machinery, New York, NY, USA, 142–153. https://doi.org/10.1145/2611286.2611290

[20] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. Flapjax: a programming language for Ajax applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. Association for Computing Machinery, New York, NY, USA, 1–20. https://doi.org/10.1145/1640089.1640091

[21] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014. REScala: bridging between object-oriented and functional style in reactive applications. In *Proceedings of the 13th International Conference on Modularity* (Lugano, Switzerland) *(MODULARITY '14)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/2577080.2577083

[22] Mingwei Samuel. 2021. *Hydroflow: A Model and Runtime for Distributed Systems Programming*. Master's thesis. EECS Department, University of California, Berkeley.

[23] 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.

[24] Meteor Software. 2024. *Meteor Framework*. https://www.meteor.com/ Online, accessed on 2024-04-21.

[25] Meteor Software. 2024. *Meteor React Tutorial*. https://react-tutorial.meteor.com/ Online, accessed on 2024-07-18.

[26] Jacqueline Xu. 2017. *Online migrations at scale*. Stripe. https://stripe.com/blog/online-migrations Online, accessed on 2024-07-18.

[27] Zhiyuan Zhan, M. Ahamad, and M. Raynal. 2005. Mixed Consistency Model: Meeting Data Sharing Needs of Heterogeneous Users. In *25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*. 209–218. https://doi.org/10.1109/ICDCS.2005.49

Received 2024-04-25; accepted 2024-08-08