

# Ur/Web: A Simple Model for Programming the Web

By Adam Chlipala

## Abstract

**The World Wide Web has evolved gradually from a document delivery platform to an architecture for distributed programming. This largely unplanned evolution is apparent in the set of interconnected languages and protocols that any Web application must manage. This paper presents Ur/Web, a domain-specific, statically typed functional programming language with a much simpler model for programming modern Web applications. Ur/Web's model is *unified*, where programs in a single programming language are compiled to other "Web standards" languages as needed; supports novel kinds of *encapsulation* of Web-specific state; and exposes *simple concurrency*, where programmers can reason about distributed, multithreaded applications via a mix of transactions and cooperative preemption. We give a tutorial introduction to the main features of Ur/Web.**

## 1. INTRODUCTION

The World Wide Web is a very popular platform today for programming certain kinds of distributed applications with graphical user interfaces (GUIs). Today's complex ecosystem of "Web standards" was not planned monolithically. Rather, it evolved gradually, from the starting point of the Web as a delivery system for static documents. The result is not surprising: there are many pain points in implementing rich functionality on top of the particular languages that browsers and servers speak. At a minimum, today's rich applications must generate HTML, for document structure; CSS, for document formatting; JavaScript, a scripting language for client-side interactivity; and messages of HTTP, a protocol for sending all of the above and more, to and from browsers. Most recent, popular applications also rely on languages like JSON for serializing complex datatypes for network communication, and on languages or APIs like SQL for storing persistent, structured data on servers. Code fragments in these different languages are often embedded within each other in complex ways, and the popular Web development tools provide little help in catching inconsistencies.

These complaints are not new, nor are language-based solutions. The Links project<sup>4,8</sup> pioneered the "tierless programming" approach, combining all the pieces of dynamic Web applications within one statically typed functional programming language. Similar benefits are attained in more recent mainstream designs, such as Google's Web Toolkit<sup>a</sup> and Closure<sup>b</sup> systems, for adding compilation on top of Web-standard languages; and Microsoft's LINQ,<sup>12</sup> for type-safe querying (to SQL databases and more) within general-purpose languages.

Such established systems provide substantial benefits to Web programmers, but there is more we could ask for. This article focuses on a language design that advances the state of the art by addressing two key desiderata. First, we bring *encapsulation* to rich Web applications, supporting program modules that treat key pieces of Web applications as private state. Second, we expose a *simple concurrency* model to programmers, while supporting the kinds of nontrivial communication between clients and servers that today's applications take advantage of. Most Web programmers seem unaware of either property as something that might be worth asking for, so part of our mission here is to evangelize for them.

We present the *Ur/Web* programming language, an extension of the Ur language,<sup>5</sup> a statically typed functional language inspired by dependent type theory. Open-source implementations of Ur/Web have been available since 2006, and several production Web applications use the language, including at least one profitable commercial site.

Ur/Web reduces the nest of Web standards to a simple programming model, coming close to retaining just the essence of the Web as an application platform, from the standpoints of security and performance. We have a single distributed system, with one server under the programmer's control and many clients that are not. The server and clients communicate only through various strongly typed communication channels, and every such interaction occurs as part of a transaction that appears to execute atomically, with no interference from other actions taking place at the same time. The server has access to persistent state in an SQL database, which is also accessed only through channels with strong types specific to the data schema. Clients maintain their GUIs through a novel variant of functional-reactive programming.

The next section expands on these points with a tutorial introduction to Ur/Web. We highlight the impact on the language design of our goals to support *encapsulation* and *simple concurrency*. Afterward, we compare with other research projects and widely used frameworks.

The open-source implementation of Ur/Web is available at:

<http://www.impredicative.com/ur/>.

## 2. A TUTORIAL INTRODUCTION TO UR/WEB

We will introduce the key features of Ur/Web through a series of refinements of one example, a multiuser chat application. Visitors to the site choose from a selection of chat

The original version of this paper was published in the *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming*. ACM, New York, NY, 2015, 153–165.

<sup>a</sup> <http://www.gwtproject.org/>.

<sup>b</sup> <https://developers.google.com/closure/>.

rooms, each of which maintains a log of messages. Any visitor to a chat room may append any line of text to the log, and there should be some way for other users to stay up-to-date on log additions. We start with a simple implementation, in the style of 20th century Web applications, before it became common to do significant client-side scripting. We evolve toward a version with instant updating upon all message additions, where a chat room runs within a single HTML page updated incrementally by client-side code. Along the way, we highlight our running themes of *encapsulation* and *simple concurrency*.

The examples from this section are written to be understandable to readers with different levels of familiarity with statically typed functional languages. Though the code should be understandable to all at a high level, some remarks (safe to skip) may require more familiarity.

## 2.1. HTML and SQL

Mainstream modern Web applications manipulate code in many different languages and protocols. Ur/Web hides most of them within a unified programming model, but we decided to expose two languages explicitly: *HTML*, for describing the structure of Web pages as trees, and *SQL*, for accessing a persistent relational database on the server. In contrast to mainstream practice, Ur/Web represents code fragments in these languages as first-class, strongly typed values.

Figure 1 gives our first chat-room implementation, relying on embedding of HTML and SQL code. While, in general, Ur/Web programs contain code that runs on both server and clients, all code from this figure runs on the server, where we are able to enforce that it is run exactly as written in the source code.

The first two lines show declarations of SQL tables, which can be thought of as mutable global variables of type “multiset of records.” Table `room`’s records contain integer IDs and string titles, while table `message`’s records contain integer room IDs, timestamps, and string messages. The former table represents the set of available chat rooms, while the latter represents the set of all (timestamped) messages sent to all rooms.

We direct the reader’s attention now to the declaration of the `main` function, near the end of Figure 1. Here we see Ur/Web’s syntax extensions for embedded SQL and HTML code. Such notation is desugared into calls to constructors of abstract syntax tree types. The `main` definition demonstrates two notations for “antiquoting,” or inserting Ur code within a quoted code fragment. The notation `{e}` asks to evaluate expression `e` to produce a subfragment to be inserted at that point, and notation `{[e]}` adds a further stage of formatting `e` as a literal of the embedded language (using type classes<sup>17</sup> as in Haskell’s `show`). Note that we are *not* exposing syntax trees to the programmer as strings, so neither antiquoting form presents any danger of *code injection attacks*, where we accidentally interpret user input as code.

What exactly does the `main` definition do? First, we run an SQL query to list all chat rooms. In our tutorial examples, we will call a variety of functions from Ur/Web’s standard library, especially various higher-order functions for using SQL query results. Such functions are *higher-order* in the

Figure 1. A simple chat-room application.

```
table room : { Id : int, Title : string }
table message : { Room : int, When : time,
                  Text : string }

fun chat id =
  let
    fun say r =
      dml (INSERT INTO message (Room, When, Text)
          VALUES ([{id}], CURRENT_TIMESTAMP, {[r.Text]}));
      chat id
  in
    title <- oneRowEl (SELECT (room.Title) FROM room
                      WHERE room.Id = {[id]});
    log <- queryX1 (SELECT message.Text FROM message
                  WHERE message.Room = {[id]}
                  ORDER BY message.When)
              (fn r => <xml>{[r.Text]}<br/></xml>);
    return <xml><body>
      <h1>Chat Room: {[title]}</h1>

      <form>
        Add message: <textbox{#Text}/>
        <submit value="Add" action={say}/>
      </form>

      <hr/>

      {log}
    </body></xml>
  end

fun main () =
  rooms <- queryX1 (SELECT * FROM room
                    ORDER BY room.Title)
                (fn r => <xml><li><a link={chat r.Id}>
                      {[r.Title]}</a></li></xml>);
  return <xml><body>
    <h1>List of Rooms</h1>

    {rooms}
  </body></xml>
```

sense that they take other functions as arguments, and we often write those function arguments *anonymously* using the syntax `fn x => e`, which defines a function that, when called, returns the value of expression `e` where parameter variable `x` is replaced with the actual argument value. We adopt a typographic convention for documenting each library function briefly, starting with `queryX1`, used in `main`:

`queryX1` Run an SQL query that returns columns from a single table (leading to the `1` in the identifier), calling an argument function on every result row. Since just a single table is involved, the input to the argument function is a record with one field per column returned by the query. The argument function should return XML fragments (leading to the `X` in the identifier), and all such fragments are concatenated together, in order, to form the result of `queryX1`.

Ur/Web follows functional languages like Haskell in enforcing *purity*, where expressions may not cause side effects. We allow imperative effects on an “opt-in” basis, with types delineating boundaries between effectful and pure code, following Haskell’s technique of monadic IO.<sup>14</sup> For instance, the `main` function here inhabits a distinguished monad for input-output. Thus, we use the `<-` notation to run an effectful computation and bind its result to a variable, and we call the `return` function to lift pure values into trivial

computations. Readers unfamiliar with the Haskell style may generally read `<-` as simple variable assignment and return in its usual meaning from C-like languages.

The remaining interesting aspect of `main` is in its use of an HTML `<a>` tag to generate a hyperlink. Instead of denoting a link via a URL as in standard HTML, we use a `link` attribute that accepts a *suspended Ur/Web remote function call*. In this case, we call `chat`, which is defined earlier. The Ur/Web implementation handles proper marshalling of arguments in suspended calls.

Now let us examine the implementation of the `chat` function, providing a page for viewing the current message log of a chat room. First, there is a nested definition of a function `say`, which will be called to append a message to the log.

`dml` Run a piece of SQL code for its side effect of mutating the database. The function name refers to SQL's data manipulation language.

This particular invocation of `dml` inserts a new row into the message table with the current timestamp, after which we trigger the logic of the main `chat` page to generate output. Note that `say`, like all remotely callable functions, appears to execute *atomically*, so the programmer need not worry about interleavings between concurrent operations by different clients.

The main body of `chat` runs appropriate queries to retrieve the room name and the full, sorted message log.

`oneRowE1` Run an SQL query that should return just one result row containing just a single column (justifying the 1) that is computed using an arbitrary SQL expression (justifying the E). That one result value becomes the result of the `oneRowE1` call.

We antiquote the query results into the returned page in an unsurprising way. The only new feature involves HTML forms. In general, we tag each input widget with a *record field name*, and then the submit button of the form includes, in its `action` attribute, an Ur/Web function that should be called upon submission, on a *record built by combining the values of all the input widgets*. We will not say any more about HTML forms, which to some extent represent a legacy aspect of HTML that has been superseded by client-side scripting. Old-style forms need to use a rigid language (HTML) for describing how to combine the values of different widgets into one request to send to the server, while these days it is more common to use a Turing-complete language (JavaScript) for the same task.

Compiling an application to run on the real Web platform requires exposing remotely callable functions (like `main`, `chat`, and `say`) via URLs. Ur/Web automatically generates pleasing URL schemes by *serializing the function-call expressions* that appear in places like `link` attributes. For instance, the link to `chat` in the declaration of `main` is compiled to a URL `/chat/NN`, where `NN` is a textual representation of the room ID.

**Adding more encapsulation.** The application in Figure 1 is rather monolithic. The database state is exposed without restrictions to all parts of the application. We would not tolerate such a lack of encapsulation in a large traditional application. Chunks of functionality should be modularized,

for example, into classes implementing data structures. The database tables here are effectively data structures, so why not try to encapsulate them as well?

The answer is that, as far as we are aware, no prior language designs allow it! As we wrote above, the general model is that the SQL database is a preexisting resource, and any part of the application may create an interface to any part of the database. We analogize such a scheme to an object-oriented language where all class fields are public; it forecloses on some very useful styles of modular reasoning. It is important that modules be able to *create their own private database tables*, without requiring any changes to other source code, application configuration files, etc., for the same reason that we do not want client code of a dictionary class to change, when the dictionary switches to being implemented with hash tables instead of search trees.

Figure 2 shows a refactoring of our application code, to present the chat-room table as a mutable abstract data type. We use Ur/Web's module system, which is in the ML tradition.<sup>11</sup> We have *modules* that implement *signatures*, which may impose information hiding by not exposing some members or by making some types *abstract*. Figure 2 defines a module `Room` encapsulating all database access.

The signature of `Room` appears bracketed between keywords `sig` and `end`. We expose an abstract type `id` of chat-room identifiers. Ur/Web code in other program modules will not be able to take advantage of the fact that `id` is really `int`, and thus cannot fabricate new IDs out of thin air. The signature exposes two methods: `rooms`, to list the IDs and titles of all chat rooms; and `chat`, exactly the remotely callable function we wrote before, but typed in terms of the abstract type `id`. Each method's type uses the transaction monad, which is like Haskell's IO monad, but with support for executing all side effects *atomically* in a remote call.

The implementation of `Room` is mostly just a copying-and-pasting of the bulk of the code from Figure 1. We only need to add a simple implementation of the `rooms` method.

`queryL1` Return as a list (justifying the L) the results of a query that only returns columns of one table (justifying the 1).

**Figure 2. A modular factorization of Figure 1.**

```
structure Room : sig
  type id
  val rooms : transaction (list {Id : id,
                                Title : string})
  val chat : id -> transaction page
end = struct
  (* ...copies of old definitions of room, message,
    and chat... *)

  val rooms = queryL1 (SELECT * FROM room
                       ORDER BY room.Title)
end

fun main () =
  rooms <- Room.rooms;
  return <xml><body>
    <h1>List of Rooms</h1>

    {List.mapX (fn r =>
      <xml><li><a link={Room.chat r.Id}>
        {[r.Title]}</a></li></xml>) rooms}
  </body></xml>
```

`List.mapX` Apply an XML-producing function to each element of a list, then concatenate together the resulting XML fragments to compute the result of `mapX`.

The code for `main` changes so that it calls methods of `Room`, instead of inlining database access.

This sort of separation of a data model is often implemented as part of the “model-view-controller” pattern. To our knowledge, that pattern had not previously been combined with guaranteed encapsulation of the associated database tables. It also seems to be novel to apply ML-style type abstraction to database results, as in our use of an `id` type here. We hope this example has helped convey one basic take-away message: giving first-class status to key pieces of Web applications makes it easy to apply standard language-based encapsulation mechanisms.

## 2.2. Client-side GUI scripting

We will develop two more variations of the chat-room application. Our modifications will be confined to the implementation of the `Room` module. Its signature is already sufficient to enable our experiments, and we will keep the same `main` function code for the rest of the tutorial.

Our first extension takes advantage of *client-side scripting* to make applications more responsive, without the need to load a completely fresh page after every user action. Mainstream Web applications are scripted with JavaScript, but, as in Links and similar languages, Ur/Web scripting is done in the language itself, which is compiled to JavaScript as needed.

**Reactive GUIs.** Ur/Web GUI programming follows the *functional-reactive* style. That is, to a large extent, the visible page is described as a transformation over *streams* (sequences of values over time). User inputs like keypresses are modeled as primitive streams, which together should be transformed into the stream of page contents that the user sees. Languages like Flapjax<sup>13</sup> and Elm<sup>9</sup> adopt the stream metaphor literally. Ur/Web follows a less pure style, where we retain the *event callbacks* of imperative programming. These callbacks modify *data sources*, which are a special kind of mutable reference cells. The only primitive streams are effectively *the sequences of values that data sources take on*, where new entries are pushed into the streams mostly via imperative code in callbacks. Both flavors of the functional-reactive style provide superior modularity compared to the standard Web model, which involves imperative mutation of the document tree, treated as a public global variable.

As a basic orientation to the concepts of sources and streams in Ur/Web, here is their type signature. We rely on two abstract parameterized types: `source  $\alpha$`  is a data source that can store values of type  $\alpha$ , and `signal  $\alpha$`  is a time-varying value that, at any particular time, has some value of type  $\alpha$ . As types and values occupy different namespaces, we often reuse an identifier (e.g., `source`) to stand for both a type and the run-time operation for allocating one of its instances, much as happens with classes and their constructors in Java. We write  $\forall \alpha. T$  for a type that is *polymorphic* in some arbitrary type parameter  $\alpha$ , much as we would write a method prototype like `<a> T f ( . . . )` in Java to bind `a` for use in `T`.

```
source :  $\forall \alpha. \text{transaction } (\text{source } \alpha)$ 
get    :  $\forall \alpha. \text{source } \alpha \rightarrow \text{transaction } \alpha$ 
set    :  $\forall \alpha. \text{source } \alpha \rightarrow \alpha \rightarrow \text{transaction } \{\}$ 

s_m    : monad signal
signal :  $\forall \alpha. \text{source } \alpha \rightarrow \text{signal } \alpha$ 
```

That is, data sources have operations for allocating them and reading or writing their values. All such operations live inside the `transaction` monad for imperative side effects. Signals, or time-varying values, happen to form a monad with appropriate operations (e.g., support `return` and the `<-` operator), as indicated by the presence of a first-class dictionary `s_m` witnessing their monadhood. One more key operation with signals is producing them from sources, via the value-level `signal` function, which turns a source into a stream that documents changes to the source’s contents.

Figure 3 demonstrates these constructs in a module implementing a GUI widget for append-only logs. The module signature declares an abstract type `t` of logs. We have methods `create`, to allocate a new log; `append`, to add a new string to the end of a log; and `render`, to produce the HTML representing a log. The type of `render` may be deceptive in its simplicity; Ur/Web HTML values (as in the `xbody` type of HTML that fits in a document body) actually are all implicitly parameterized in the dataflow style, and they are equipped to rerender themselves after changes to the data sources they depend on.

Figure 3. Append-only log module for GUIs.

```
structure Log : sig
  type t
  val create : transaction t
  val append : t -> string -> transaction {}
  val render : t -> xbody
end = struct
  datatype log =
    Nil
  | Cons of string * source log

  type t = {Head : source log,
            Tail : source (source log)}

  val create =
    s <- source Nil;
    s' <- source s;
    return {Head = s, Tail = s'}

  fun append t text =
    s <- source Nil;
    oldTail <- get t.Tail;
    set oldTail (Cons (text, s));
    set t.Tail s;

    log <- get t.Head;
    case log of
      Nil => set t.Head (Cons (text, s))
    | _ => return ()

  fun render_aux log =
    case log of
      Nil => <xml></xml>
    | Cons (text, rest) => <xml>
      {[text]}<br/>
      <dyn signal={log <- signal rest;
                    return (render_aux log)}>/>
    </xml>

  fun render t = <xml>
    <dyn signal={log <- signal t.Head;
                  return (render_aux log)}>/>
  </xml>
end
```



The workhorse data structure of logs is an *algebraic datatype* `log`. Algebraic datatypes are the classic tool for structured data in statically typed functional programming, and they behave similarly to *tagged unions* in some other languages (or Scala’s case classes). The definition of `log` gives an exhaustive list of the *constructors* for logs (`Nil` and `Cons`) with the types of arguments that they take, and later we use case expressions to deconstruct such values, giving different code to evaluate depending on which constructor built a value. Our definition of `log` looks almost like a standard definition of lists of strings. The difference is that the tail of a nonempty `log` has type `source log`, rather than just `log`. We effectively have a type of lists that supports imperative replacement of list tails, but not replacement of heads.

The type `t` of logs is a record of two fields. Field `Head` is a modifiable reference to the current log state, and `Tail` is a “pointer to a pointer,” telling us which source cell we should overwrite next to append to the list. Methods `create` and `append` involve a bit of intricacy to update these fields properly, but we will not dwell on the details. We only mention that the point of this complexity is to avoid rerendering the whole log each time an entry is appended; instead, only a constant amount of work is done per append, to modify the document tree at the end of the log. That sort of pattern is difficult to implement with pure functional-reactive programming.

The most interesting method definition is for `render`. Our prior examples only showed building HTML fragments with no dependencies on data sources. We create dependencies via a pseudotag called `<dyn>`. The `signal` attribute of this tag accepts a signal, a time-varying value telling us what content should be displayed at this point on the page at different times. Crucially, the `signal` monad rules out imperative side effects, instead capturing pure dataflow programming. Since it is a monad, we have access to the usual monad operations `<-` and `return`, in addition to the `signal` function for lifting sources into signals.

Let us first examine the definition of `render_aux`, a recursive helper function for `render`. The type of `render_aux` is `log -> xbody`, displaying a log as HTML. Empty logs are displayed as empty HTML fragments, and nonempty logs are rendered with their textual heads followed by recursive renderings of their tails. (Following functional-programming tradition, we write “head” and “tail” respectively for the first element of a list and the remainder of the list minus that element.) However, the recursive call to `render_aux` is not direct. Instead it appears inside a `<dyn>` pseudotag. We indicate that this subpage depends on the current value of the tail (a data source), giving the computation to translate from the tail’s value to HTML. Now it is easy to define `render`, mostly just duplicating the last part of `render_aux`.

An important property of this module definition is that a rendered log automatically updates in the browser after every call to `append`, even though we have not coded any explicit coupling between these methods. The Ur/Web runtime system takes care of the details, once we express GUIs via parameterized dataflow.

Client code may use logs without knowing implementation details. In the standard model for programming

browser GUIs with JavaScript, mistakes in one code module may wreck subtrees that other modules believe they control. For instance, subtrees are often looked up by string ID, creating the possibility for two different libraries to choose the same ID unwittingly for different subtrees. With the Ur/Web model, the author of module `Log` may think of it as *owning* particular subtrees of the HTML document as private state. Standard module encapsulation protects the underlying data sources from direct modification by other modules, and rendered logs only have dataflow dependencies on the values of those sources.

**Remote procedure calls.** The GUI widget for displaying the chat log is only one half of the story if we are to write an application that runs within a single page. We also need a way for this application to contact the server, to trigger state modifications and receive updated information. Ur/Web’s first solution to that problem is *remote procedure calls* (RPCs), allowing client code to run particular function calls as if they were executing on the server, with access to shared database state. Client code only needs to wrap such calls to remotely callable functions within the `rpc` keyword, and the Ur/Web implementation takes care of all network communication and marshalling. Every RPC appears to execute *atomically*, just as for other kinds of remote calls.

Figure 4 reimplements the `Room` module to take advantage of RPCs and the `Log` widget.

`List.foldl` As in ML, step through a list, applying a function `f` to each element, so that, given an initial accumulator `a` and a list `[x1, . . . , xn]`, the result is `f(xn, . . . , f(x1, a) . . .)`.

`List.app` Apply an effectful function to every element of a list, in order.

The code actually contains few new complexities. Our basic strategy is for each client to maintain the timestamp of the *most recent* chat message it has received. The textbox for user input is associated with a freshly allocated `source string`, via the `<ctextbox>` pseudotag (“c” is for “client-side scripting”). Whenever the user modifies the text shown in this box, the associated source is automatically mutated to contain the latest text. When the user clicks a button to send a message, we run the callback code in the button’s `onclick` attribute, *on the client*, whereas the code for this example outside of `on*` attributes runs on the server. This code makes an RPC, telling the server both the new message text and the last timestamp that the client knows about. The server sends back a list of all chat messages newer than that timestamp, and client code iterates over that list, adding each message to the log; and then updates the last-seen timestamp accordingly, taking advantage of the fact that the RPC result list will never be empty, as it always contains at least the message that this client just sent. An `onload` event handler in the `body` tag initialized the log in the first place, appending each entry returned by an initial database query.

Note how seamless the use of the `Log` module is. We allocate a new log, drop its rendering into the right part of the page, and periodically append to it. Pure functional-reactive programming would require some acrobatics to interleave the event streams generated as input to the log system, from the two syntactically distinct calls to `Log.append`.

**Figure 4. A client-code-heavy chat-room application.**

```

structure Room : sig
  type id
  val rooms : transaction (list {Id : id,
                                Title : string})

  val chat : id -> transaction page
end = struct
  table room : { Id : int, Title : string }
  table message : { Room : int, When : time,
                   Text : string }
  val rooms = queryL1 (SELECT * FROM room
                      ORDER BY room.Title)

  (* New code w.r.t. Figure 2 starts here. *)

  fun chat id =
    let
      fun say text lastSeen =
        dml (INSERT INTO message (Room, When, Text)
            VALUES ([id], CURRENT_TIMESTAMP, [text]));
        queryL1 (SELECT message.Text, message.When
                FROM message
                WHERE message.Room = {[id]}
                AND message.When > {[lastSeen]}
                ORDER BY message.When DESC)

      val maxTimestamp =
        List.foldl (fn r acc => max r.When acc) minTime
    in
      title <- oneRowEl (SELECT (room.Title) FROM room
                        WHERE room.Id = {[id]});
      initial <- queryL1 (SELECT message.Text,
                          message.When
                          FROM message
                          WHERE message.Room = {[id]}
                          ORDER BY message.When DESC);

      text <- source "";
      log <- Log.create;
      lastSeen <- source (maxTimestamp initial);

      return <xml><body onload={
        List.app (fn r => Log.append log r.Text) initial>
        <h1>Chat Room: {[title]}</h1>

        Add message: <textbox source={text}/>
        <button value="Add" onclick={fn _ =>
          txt <- get text;
          set text "";
          lastSn <- get lastSeen;
          newMsgs <- rpc (say txt lastSn);
          set lastSeen (maxTimestamp newMsgs);
          List.app (fn r => Log.append log r.Text)
            newMsgs}/>
        <hr/>
        {Log.render log}
      }</body></xml>
    end
  end

```

## 2.3. Message passing from server to client

Web browsers make it natural for clients to contact servers via HTTP requests, but the other communication direction may also be useful. One example is our chat application, where only the server knows when a client has posted a new message, and we would like the server to notify all other clients in the same chat room. Ur/Web presents an abstraction where servers are able to send typed *messages* directly to clients, and the Ur/Web compiler and runtime system implement this abstraction once and for all on top of standard protocols and APIs.

The messaging abstraction is influenced by concurrent programming languages like Erlang<sup>1</sup> and Concurrent ML.<sup>15</sup> Communication happens over unidirectional *channels*. Every channel has an associated client and a type. The server may *send* any value of that type to the channel, which conceptually adds the message to a queue on the client. Clients

asynchronously *receive* messages from channels for which they have handles, conceptually dequeuing from local queues, blocking when queues are empty. Any remote call may trigger any number of sends to any number of channels. All sends in a single remote call appear to take place *atomically*.

The API for channels is straightforward:

```

channel : ∀α. transaction (channel α)
recv    : ∀α. channel α → transaction α
send    : ∀α. channel α → α → transaction {}

```

Figure 5 gives another reimplementing of Room, this time using channels to keep clients synchronized at all times, modulo small amounts of lag. We retain the same room and message tables as before, but we also add a new table subscriber, tracking which clients are listening for notifications about which rooms. (Thanks to Ur/Web's approach to encapsulation of database tables, we need not change any other source or configuration files just because we add a new private table.) Every row of subscriber has a room ID Room and a channel Chan that is able to receive strings.

Now the chat method begins by allocating a fresh channel with the channel operation, which we immediately insert into subscriber. Compared to Figure 4, we drop the client-side timestamp tracking. Instead, the server will use channels to notify all clients in a room, each time a new message is posted there. In particular, see the tweaked definition of say.

**queryL1** Run an SQL query returning columns from just a single table (justifying the 1), applying a function to each result in order, solely for its imperative side effects (justifying the  $\mathbb{I}$ ).

We loop over all channels associated with the current room, sending the new message to each one.

There is one last change from Figure 4. The onload attribute of our <body> tag still contains code to run immediately after the page is loaded. This time, before we initialize the Log structure, we also create a new thread with the spawn primitive. That thread loops forever, blocking to receive messages from the freshly created channel and add them to the log.

Threads follow a simple *cooperative* semantics, where the programming model says that, at any moment in time, at most one thread is running *across all clients of the application*. Execution only switches to another thread when the current one terminates or executes a blocking operation, among which we have RPCs and channel *recv*. Of course, the Ur/Web implementation will run many threads at once, with an arbitrary number on the server and one JavaScript thread per client, but the implementation ensures that no behaviors occur that could not also be simulated with the simpler one-thread-at-a-time model.

This simple approach has pleasant consequences for program modularity. The example of Figure 5 only shows a single program module taking advantage of channels. It is possible for channels to be used freely throughout a program, and the Ur/Web implementation takes care of routing messages to clients, while maintaining the simple thread semantics.

Figure 5 contains no explicit deallocation of clients that have stopped participating. The Ur/Web implementation detects client departure using a heartbeating mechanism.

**Figure 5. Final chat-room application.**

```
structure Room : sig
  type id
  val rooms : transaction (list {Id : id,
                                Title : string})

  val chat : id -> transaction page
end = struct
  table room : { Id : int, Title : string }
  table message : { Room : int, When : time,
                   Text : string }
  val rooms = queryL1 (SELECT * FROM room
                      ORDER BY room.Title)

  (* New code w.r.t. Figure 2 starts here. *)

  table subscriber : { Room : int,
                      Chan : channel string }

  fun chat id =
    let
      fun say text =
        dml (INSERT INTO message (Room, When, Text)
            VALUES ([[id]], CURRENT_TIMESTAMP, [[text]]));
        queryL1 (SELECT subscriber.Chan FROM subscriber
                WHERE subscriber.Room = [[id]])
                (fn r => send r.Chan text)
    in
      chan <- channel;
      dml (INSERT INTO subscriber (Room, Chan)
          VALUES ([[id]], [[chan]]));

      title <- oneRowE1 (SELECT (room.Title) FROM room
                        WHERE room.Id = [[id]]);
      initial <- queryL1 (SELECT message.Text,
                          message.When
                        FROM message
                        WHERE message.Room = [[id]]
                        ORDER BY message.When DESC);

      text <- source "";
      log <- Log.create;

      return <xml><body onload={
        let
          fun listener () =
            text <- recv chan;
            Log.append log text;
            listener ()
        in
          spawn (listener ());
          List.app (fn r => Log.append log r.Text) initial
        end}>
        <h1>Chat Room: {[[title]]}</h1>

        Add message: <ctextbox source={text}/>
        <button value="Add" onclick={fn _ =>
          txt <- get text;
          set text "";
          rpc (say txt)}/>

        <hr/>

        {Log.render log}
      </body></xml>
    end
  end
```

When a client departs, the runtime system *atomically deletes from the database all references to that client's channels*, providing a kind of modularity similar to what garbage collection provides for heap-allocated objects.

### 3. RELATED WORK

The space of Web-development tools is a busy one, both in the research world and in the mainstream, so we will not have space to take a nearly complete tour of it; we provide a longer comparison elsewhere.<sup>7</sup>

A variety of research programming languages and libraries were under development in the early 2000s, in that interesting transitional period where the Web had become mainstream but the highly interactive “AJAX” style of Gmail and so on had not yet become common. The PLT Scheme Web Server<sup>10</sup> provides completely first-class support for URLs as first-class functions (more specifically, continuations), making it very easy to construct many useful abstractions. The Links<sup>8</sup> language pioneered strong static checking of multiple-tier Web applications, where the code for all tiers is collected in a single language. Hop<sup>16</sup> is another unified Web programming language, but is dynamically typed and based on Scheme. Ocsigen<sup>2,3</sup> is an OCaml-based platform for building dynamic Web sites in a unified language, with static typing that rules out many potential programming mistakes. The Opa language<sup>c</sup> is another statically typed unified language for database-backed Web applications.

In broad strokes, how does Ur/Web compare to these others from the languages research community? We believe the key comparison points relate to those features of Ur/Web that we have stressed throughout this article: encapsulation and concurrency. No other languages allow enforced encapsulation of database state inside of modules, and the languages listed above all support client-side GUIs only through mutation of a global document tree, with no enforced encapsulation of client-side state pieces. Their semantics for concurrency are substantially more complicated than Ur/Web's transactions.

Several other languages and frameworks support functional-reactive programming for client-side Web GUIs, including Flapjax,<sup>13</sup> which is available in one flavor as a JavaScript library; and Elm,<sup>9</sup> a new programming language. These libraries implement the original, “pure” version of functional-reactive programming, where key parts of programs are written as pure functions that transform input streams into streams of visible GUI content. Such a style is elegant in many cases, but it does not seem compatible with the modularity patterns we demonstrated in Section 2.2, where it is natural to spread input sources to a single stream across different parts of a program. Ur/Web supports that kind of modularity by adopting a hybrid model, with imperative event callbacks that trigger recomputation of pure code.

As far as we are aware, Ur/Web was the first Web programming tool to support impure functional-reactive programming, but the idea of reactive GUI programming in JavaScript is now mainstream, and too many frameworks exist to detail here.

One popular JavaScript framework is Meteor,<sup>d</sup> distinguished by its support for a particular reactive programming style. It integrates well with mainstream Web development tools and libraries, which is a nontrivial advantage for most programmers. Its standard database support is for MongoDB, with no transactional abstraction or other way of taming simultaneous complex state updates. Like Opa, Meteor allows modules to encapsulate named database elements, but an exception is thrown if two modules

<sup>c</sup> <http://opalang.org/>.

<sup>d</sup> <http://www.meteor.com/>.



have chosen the same string name for their elements; module authors must coordinate on how to divide a global namespace. Meteor supports automatic publishing of server-side database changes into client-side caches, and then from those caches into rendered pages. In addition to automatic updating of pages based on state changes, Meteor provides a standard DOM-based API for walking document structure and making changes imperatively, though it is not very idiomatic. Meteor's machinery for reactive page updating involves a more complex API than Ur/Web's. Its central concept is of *imperative* functions that need to be rerun when any of their dependencies change, whereas Ur/Web describes reactive computations in terms of *pure* code within the signal monad, such that it is easy to rerun only *part* of a computation, when not all of its dependencies have changed. Forcing purity on these computations helps avoid the confusing consequences of genuine side effects being repeated on each change to dependencies. The five lines of code near the start of Section 2.2, together with the `<dyn>` pseudotag, give the complete interface for reactive programming in Ur/Web, in contrast with tens of pages of documentation (of dynamically typed functions) for Meteor.

Other popular JavaScript frameworks include Angular.js,<sup>e</sup> Backbone,<sup>f</sup> Ractive,<sup>g</sup> and React.<sup>h</sup> A commonality among these libraries seems to be heavyweight approaches to the basic structure of reactive GUIs, with built-in mandatory concepts of models, views, controllers, templates, components, etc. In contrast, Ur/Web has its 5-line API of sources and signals. These mainstream JavaScript frameworks tend to force elements of reactive state to be enumerated explicitly as fields of some distinguished object, instead of allowing data sources to be allocated dynamically throughout the modules of a program and kept as private state of those modules.

#### 4. CONCLUSION

We have presented the design of Ur/Web, a programming language for Web applications, focusing on a few language-design ideas that apply broadly to a class of distributed applications. Our main mission is to promote two desiderata that programmers should be asking for in their Web frameworks, but which seem almost absent from mainstream discussion: stronger and domain-specific modes of *encapsulation* and *simple concurrency* models based on transactions.

Ur/Web is used in production today for a number of applications, including at least one<sup>i</sup> with thousands of paying customers. We list more examples elsewhere.<sup>7</sup> We have also written elsewhere about Ur/Web's whole-program optimizing compiler,<sup>6</sup> which has placed favorably in a third-party Web-framework benchmarking initiative,<sup>j</sup> for instance achieving the second-best throughput (out of about 150 participating frameworks) of about 300,000 requests per second on the test closest to a full Web app.

<sup>e</sup> <https://angularjs.org/>.

<sup>f</sup> <http://backbonejs.org/>.


<sup>g</sup> <http://www.ractivejs.org/>.

<sup>h</sup> <http://facebook.github.io/react/>.

<sup>i</sup> <http://www.bazqux.com/>.

<sup>j</sup> <http://www.techempower.com/benchmarks/>.

#### Acknowledgments

This work has been supported in part by National Science Foundation grant CCF-1217501. The author thanks Christian J. Bell, Benjamin Delaware, Xavier Leroy, Clément Pit-Claudel, Benjamin Sherman, and Peng Wang for their feedback on drafts. 

#### References

1. Armstrong, J. Erlang – A survey of the language and its industrial applications. In *Proceedings of the Symposium on Industrial Applications of Prolog (INAP96)* (1996), 16–18.
2. Balat, V. Ocsigen: Typing Web interaction with Objective Caml. In *Proceedings of the 2006 Workshop on ML*, ML '06 (New York, NY, USA, 2006). ACM, 84–94.
3. Balat, V., Vouillon, J., Yakobowski, B. Experience report: Ocsigen, a Web programming framework. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09 (New York, NY, USA, 2009). ACM, 311–316.
4. Cheney, J., Lindley, S., Wadler, P. A practical theory of language-integrated query. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13 (New York, NY, USA, 2013). ACM, 403–416.
5. Chlipala, A. Ur: Statically-typed metaprogramming with type-level record computation. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10 (New York, NY, USA, 2010). ACM, 122–133.
6. Chlipala, A. An optimizing compiler for a purely functional web-application language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015 (New York, NY, USA, 2015). ACM, 10–21.
7. Chlipala, A. Ur/Web: A simple model for programming the Web. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15 (New York, NY, USA, 2015). ACM, 153–165.
8. Cooper, E., Lindley, S., Wadler, P., Yallop, J. Links: Web programming without tiers. In *Proceedings of the 5th International Conference on Formal Methods for Components and Objects*, FMCO'06 (Berlin, Heidelberg, 2007). Springer-Verlag, 266–296.
9. Czaplicki, E., Chong, S. Asynchronous functional reactive programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13 (New York, NY, USA, 2013). ACM, 411–422.
10. Krishnamurthi, S., Hopkins, P.W., McCarthy, J., Graunke, P.T., Pettyjohn, G., Felleisen, M. Implementation and use of the PLT Scheme Web Server. *Higher Order Symbol. Comput.* 20, 4 (2007), 431–460.
11. MacQueen, D. Modules for standard ML. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84 (New York, NY, USA, 1984). ACM, 198–207.
12. Meijer, E., Beckman, B., Bierman, G. LINQ: Reconciling objects, relations and XML in the .NET framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06 (New York, NY, USA, 2006). ACM, 706–706.
13. Meyerovich, L.A., Guha, A., Baskin, J., Cooper, G.H., Greenberg, M., Bromfield, A., Krishnamurthi, S. Flapjax: A programming language for Ajax applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09 (New York, NY, USA, 2009). ACM, 1–20.
14. Peyton Jones, S.L., Wadler, P. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93 (New York, NY, USA, 1993). ACM, 71–84.
15. Reppy, J.H. *Concurrent Programming in ML*. Cambridge University Press, New York, NY, USA, 1999.
16. Serrano, M., Galliesio, E., Loitsch, F. Hop, a language for programming the Web 2.0. In *Proceedings of the First Dynamic Languages Symposium*, DLS '06 (New York, NY, USA, 2006). ACM.
17. Wadler, P., Blott, S. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89 (New York, NY, USA, 1989). ACM, 60–76.

Adam Chlipala (adamc@csail.mit.edu), MIT CSAIL, Cambridge, MA.

Copyright held by author.  
Publication rights licensed to ACM. \$15.00



Watch the author discuss his work in this exclusive *Communications* video.  
<http://cacm.acm.org/videos/ur-web>



Copyright of Communications of the ACM is the property of Association for Computing Machinery and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.