Ur/Web: A Simple Model for Programming the Web

Adam Chlipala
MIT CSAIL
adamc@csail.mit.edu

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 and discuss the language implementation and the production Web applications that use it.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques - Modules and interfaces; D.3.2 [*Programming Languages*]: Language Classifications - Concurrent, distributed, and parallel languages; Applicative (functional) languages

Keywords Web programming languages; encapsulation; transactions; remote procedure calls; message passing; relational databases; functional-reactive programming

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 HTTP, a protocol for sending all of the above and more, to and from browsers. Most recent, popular applications also rely on a language like JSON for serializing complex datatypes for network communication, and on a language or API 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 [7, 11] pioneered the "tierless programming" approach, combining all the pieces of dynamic Web applications within one statically typed functional language. More recent designs in the mainstream reap some similar benefits, as in Google's Web Toolkit[1] and Closure[2] systems, for adding compilation on top of Web-standard languages; and Microsoft's LINQ [27], 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 paper 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 in this paper is to evangelize for them.

We present the **Ur/Web** programming language, an extension of the Ur language [8], a statically typed functional language inspired by dependent type theory. A prior paper described Ur and its type system, but did not discuss the Web-specific extensions. 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.

- An application is a program in one language (Ur/Web) that runs on one **server** and many **clients**, with automatic compilation of parts of programs into the languages appropriate to the different nodes (e.g., JavaScript). The server is completely under the programmer's control, while clients may deviate arbitrarily from code we provide to them to run. For reasons of performance scaling or reliability, we may use multiple physical machines to implement server functionality, but it is sound for the programmer to imagine all server code running in a single isolated machine.
- All objects passed between parts of the application are **strongly typed**. Applications may be written with no explicit marshaling

POPL '15, January 15–17, 2015, Mumbai, India.

ACM 978-1-4503-3300-9/15/01…$15.00.
http://dx.doi.org/10.1145/2676726.2677004

[1] `http://www.gwtproject.org/`

[2] `https://developers.google.com/closure/`

or other conversion of data between formats. Where snippets of code appear as first-class values, they are presented as abstract syntax trees, ruling out flaws like *code injection vulnerabilities* that rely on surprising consequences of concatenating code-as-strings.

- The only persistent state in the server sits in an SQL database, accessed through a strongly typed SQL interface.
- The server exposes a set of typed functions that are callable remotely. A client begins interacting with the application in a new browser tab by **making a remote procedure call to one of these functions, with arbitrary correctly typed arguments**. The server runs the associated function **atomically**, with no opportunity to observe interference by any other concurrent operations, generating an HTML page that the client displays.
- The HTML page in a client may contain traditional *links* to other pages, which are represented as **suspended calls to other remotely callable functions**, to be forced when a link is followed, to generate the new HTML page to display.
- Any HTML page may also contain **embedded Ur/Web code** that runs in the client. Such code may spawn as many client-side threads as is convenient, and the threads obey a **cooperative multithreading** semantics, where one thread runs at a time, and we only switch threads during well-defined blocking operations. Threads may modify the GUI shown to the user, via a **functional-reactive programming** system that mixes dataflow programming with imperative callbacks.
- Client-side thread code may also make blocking **remote procedure calls** treated similarly to those for regular links. Such a call may return a value of any function-free type, not just HTML pages; and the thread receiving the function result may compute with it to change the visible GUI programmatically, rather than by loading a completely new page. As before, every remote procedure call appears to execute **atomically** on the server.
- Server code may allocate **typed message-passing channels**, which may be both stored in the database and returned to clients via remote function calls. The server may send values to a channel, and a client that has been passed the channel may receive those values asynchronously. Channel sends are included in the guarantee of atomicity for remote calls on the server; all sends within a single call execution appear to transfer their messages to the associated channels atomically.

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**. Next, we describe key implementation techniques in the Ur/Web compiler and runtime system, and we evaluate the effectiveness of the language, partly through surveying deployed applications that use it.

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 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**.

```
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 <- oneRowE1 (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>
```

Figure 1. A simple chat-room application

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. We use the type system of Ur [8] to define rich syntax tree types, where the generic type system is sufficient to enforce the typing rules of the embedded languages, HTML and SQL. Going into the details of type encoding would take us off-track in this paper, so the reader may pretend that Ur/Web includes special built-in support for type-checking HTML and SQL.

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.

It is unusual for a programming language to treat SQL tables as declared within the language. The more common view is that the SQL database exists as a resource "out there somewhere," and the programming language merely connects to it. Our strange choice has important consequences for encapsulation, which we will get to shortly.

We direct the reader's attention now to the declaration of function `main`, 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, at which point the normal Ur type-checker may validate the type-correctness of embedded fragments. 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 [37] 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. 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 can be 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`.

Note that a remotely callable function like `main` lives in a distinguished monad for input-output [29] as in Haskell, since Ur is purely functional. 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.

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 marshaling of arguments in suspended calls.

Now let us examine the implementation of function `chat`, 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 **d**ata **m**anipulation **l**anguage.

This particular invocation of `dml` inserts a new row into the `message` table with the current timestamp, after which we display the same page as `chat` itself would generate. 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. To tune performance, programmers might want to indicate the boundaries of atomic execution units explicitly, but we have found that it works well to make the atomic units be precisely the remote calls, which saves programmers from needing to write any explicit code to begin or end transactions.

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 **e**xpression (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. Forms are a non-Turing-complete language for input solicitation, and these days it is more common to use a Turing-complete language (JavaScript) for the same task.

Note that this first example already displays a small bit of encapsulation: the local function `say` may only be referenced within the declaration of `chat`. Clients may ignore the intended order of link-following and visit `say` directly, but we at least know our application itself will not accidentally generate links to `say` outside the intended scope.

Compiling an application to run on the real Web platform requires exposes 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. The link to `say` within the `submit` button above is compiled like `/say/NN`, including the value of a local variable `id` implicitly captured in the function.

The Ur/Web compiler handles generation of URL schemes in such a way that it is guaranteed that no two functions will wind up with conflicting schemes. This approach contrasts sharply with the manual URL routing found in most mainstream frameworks. There are serious costs to modularity when composing two libraries requires understanding their URL schemes.

Some other systems, like the PLT Scheme Web Server [20] and Links [11], avoid the problem by generating non-human-readable URLs, standing for continuation references. We prefer to keep URLs readable, since they serve as a common interface with the broader Internet world. However, Links pioneered many of the other design elements on display in Figure 1, including type-safe combination of HTML and database code within a single language. One further key difference is that Ur/Web works with direct embedding of SQL code, whereas Links compiles to SQL from a subset of itself, presented as a monadic query language [7].

The key distinguishing elements of Ur/Web's design, in comparison to past languages, become clearer when we turn to taking better advantage of encapsulation. Before doing so, we pause to discuss some relative strengths and weaknesses of the HTML and SQL encodings in Ur/Web.

Ur/Web effectively encodes XML with algebraic datatypes, which are not expressive enough to capture all of the constraints associated with XML schemas. *Regular expression types* [19] are an established approach to enforcing the rules of XML more literally, and they have been integrated with ML-style languages in the OCamlDuce project [16]. Programmers may benefit from that style of more precise type-checking. On the other hand, more complex static checking of XML may be more difficult for programmers

```
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>
```

Figure 2. A modular factorization of the first application

to understand. An additional benefit of Ur/Web's approach is that XML checking need not be built into the language (as it is in, e.g., OCamlDuce) but is instead encoded as a library using Ur's rich but general-purpose type system [8]. We use row types to encode context constraints on XML elements, so that the generic Ur type-inference engine may do the rest of the work.

A similar comparison may be drawn with other approaches to encoding relational database queries. The *language-integrated query* approach represents queries within a restricted subset of the host programming language, rather than working with explicit embedded syntax of a language like SQL. Microsoft's LINQ [27] is the well-known example that gives the approach its name, but the contemporaneous Links language [7, 11] uses static type checking and term rewriting to provide more principled guarantees about which expressions will compile to SQL successfully. Ur/Web encodes SQL as a library, with no special type-inference support required in the compiler, and the programmer benefits from the explicit control over exactly which code will be passed to the database system. Performance fine-tuning may be most effective with this sort of direct approach. On the other hand, explicit manipulation of query syntax can be more tedious than writing query code directly in the host language, relying on compilation to the language of a database system; and the use of multiple languages encourages some amount of duplicate definition of library functions. Overall, on this comparison point the author is agnostic and could imagine a successor to Ur/Web being developed with language-integrated query.

2.1.1 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, e.g. 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 [23]. 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`).

`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 to call 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.

Strong encapsulation is not automatic when using conventional database engines. The database server will generally be configured to allow ad-hoc querying by humans and other applications, where the data abstraction in a particular Ur/Web application need not be respected. We imagine a future version of Ur/Web with tighter data storage integration, where by default only an application itself may access particular tables. For now, though, data abstraction for SQL tables remains useful as a way to reason about Ur/Web programs, and it is possible to associate each application with a new database user whose authentication credentials are only given to the application daemon. The Ur/Web compiler will automatically initialize the schema of a database, and each application checks on start-up that the schema matches what is present in its source code.

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

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

 fun render t = <xml>
  <dyn signal={log <- signal t.Head;
               return (render' log)}/>
 </xml>
end
```

Figure 3. A module implementing a GUI for an append-only log

languages, Ur/Web scripting is done in the language itself, which is compiled to JavaScript as needed.

2.2.1 Reactive GUIs

Ur/Web GUI programming follows the **functional-reactive** style. That is, to a large extent, the visible page is described via *dataflow*, as a pure function over some primitive streams of values. As the primitive streams evolve, the language runtime system automatically rerenders just those parts of the page that are affected. Languages like Flapjax [28] and Elm [14] adopt the stream metaphor literally, where the primitive streams represent, e.g., mouse clicks made by the user. Ur/Web adopts 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.

As a basic orientation to these concepts in Ur/Web, here is their type signature. We write :: to ascribe a *kind* to a type-level identifier, where, for instance, Type $\rightarrow$ Type indicates a *type family* parameterized over one type argument. We write : to ascribe a type to a value-level identifier. 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.

$$
\begin{array}{rcl}
\mathsf{source} & :: & \mathsf{Type} \rightarrow \mathsf{Type} \\
\mathsf{source} & : & \forall \alpha.\ \mathsf{transaction}\ (\mathsf{source}\ \alpha) \\
\mathsf{get} & : & \forall \alpha.\ \mathsf{source}\ \alpha \rightarrow \mathsf{transaction}\ \alpha \\
\mathsf{set} & : & \forall \alpha.\ \mathsf{source}\ \alpha \rightarrow \alpha \rightarrow \mathsf{transaction}\ \{\}
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{signal} & :: & \mathsf{Type} \rightarrow \mathsf{Type} \\
\mathsf{s\_m} & : & \mathsf{monad\ signal} \\
\mathsf{signal} & : & \forall \alpha.\ \mathsf{source}\ \alpha \rightarrow \mathsf{signal}\ \alpha
\end{array}
$$

That is, data sources are a polymorphic type family, with operations for allocating them and reading or writing their values. All such operations live inside the `transaction` monad, which contains imperative side effects in the manner of Haskell's IO monad [29]. *Signals*, or time-varying values, are another polymorphic type family, which happens to form a monad with appropriate operations, 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 use, 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 deceiving 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.

The workhorse data structure of logs is an algebraic datatype `log`, which 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 that are vacuously dataflow-parameterized. We create dependencies via an HTML 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'`, a recursive helper function for `render`. The type of `render'` 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. However, the recursive call to `render'` 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'`.

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. The standard Web model involves imperative mutation of the document tree, which is treated as a public global variable. 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.

Classic pure functional-reactive frameworks provide similar advantages, but it can be challenging to fit a full application into the model of pure stream transformers. In our experience, many programmers find the callback style easier to wrap their heads around. Consider the example of the `Log` module. We *could* express the log creation or rendering method as a function over a stream of requests to append entries. However, building these streams in practice may be nontrivial, forcing explicit demultiplexing of the variety of user actions and network responses that may produce log entries. (We see an example of multiple log-entry sources in the next subsection.) The usual modularity advantages of mutable state fit well in this GUI domain, and they are compatible with maintaining the dataflow part of functional-reactivity. Flapjax [28] supports the imperative model, via a method for pushing events into *any* stream (even those that we might wish client code saw as read-only), but this method is suggested for use only in interfacing with legacy, callback-based code.

2.2.2 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 marshaling. 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 $[x_1, \ldots, x_n]$, the result is $f(x_n, \ldots f(x_1, a) \ldots)$.

`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

```
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 <- 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;
   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: <ctextbox 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
```

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

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 is the use of the `Log` module. We allocate a new log, drop its rendering into the right part of the page, and peri-

```
fun main () =
 tag <- source "";
 image <- source <xml/>;

 return <xml><body>
  Tag: <ctextbox source={tag}
   onchange={tag <- get tag;
    found <- rpc (tagExists tag);
    if found then
     set image <xml>
      <img src={url (serveImage tag)}/></xml>
    else return ()}/><br/>
  <dyn signal={signal image}/>
```

Figure 5. Partial Ur/Web code for image-lookup example

odically 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`. Another challenge is fitting effectful RPCs into a pure functional-reactive framework. Elm [14] supports injection of RPCs in pure stream-transformation code, with a novel mechanism to allow local computation to continue while waiting for RPC responses. It is not clear how to extend such a technique for impure RPCs, which are useful for precisely the kind of call we use in Figure 4, to modify a shared database. For pure RPCs, we can realize similar semantics to Elm's merely by spawning RPCs in their own threads, which write to sources after RPCs complete.

For instance, consider the third example from a paper about Elm [14], where users type text into a textbox, and an image on the same page updates according to the result of an image-search query to a remote server, using the latest text. Here is an excerpt of the Elm code.

```
(inputField, tags) = Input.text "Enter a tag"

getImage tags =
  lift (fittedImage 300 200)
       (syncGet (lift requestTag tags))

... async (getImage tags) ...
```

Built-in function `syncGet` makes an HTTP request, and built-in function `async` causes evaluation to run in a separate thread. The essential dataflow action in this example is from `tags`, a stream recording successive contents of a freshly created textbox, into the details of the HTTP request to be made via `syncGet`.

We can actually duplicate the same functionality in Ur/Web, in an equally pure way, by taking advantage of the fact that the HTML `<img>` tag already has built into it the idea of contacting a remote server to retrieve an image by URL. A dynamically varying piece of HTML may simply contain different `<img>` tags at different times, and the browser handles retrieving images as usual. However, for a fairer comparison, we should consider an interaction that does require some custom logic. Figure 5 shows Ur/Web code for a variant approach, where, whenever the contents of the tag textbox change, we first ask the server by RPC if any image exists matching that tag. If none exists, we make no changes to which image is displayed. If a match is found, we change to the new image, using the built-in function `url`, which coerces a remote function call into an abstract type of URLs. In this case, the URL is for a server-side function that computes an image based on input text. All of the reactive computation happens inside an `onchange` handler for the textbox, called whenever the text changes.

The code in Figure 5 blocks the whole client-side application while an RPC runs, but that problem is easily solved by putting the whole `onchange` attribute code snippet into a new thread, with the `spawn` keyword that we say more about in the next subsection. Overall, the Ur/Web version of this functionality is moderately more complicated than the Elm version, but all of the imperative side effects can be encapsulated within a module in the style of `Log` from Figure 3, creating an observationally pure image-search widget. Some functionality, like that of the `Log` module itself, is inherently easier to orchestrate modularly with mutable state, allowing widely separated parts of a program, with their own private state, to create data flows into a shared component, with no explicit fan-in merging their different streams.

```
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]}));
    queryI1 (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
```

Figure 6. Final chat-room application

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. Clients can *poll* the server with periodic HTTP requests, guaranteeing that the lag to learn of a new message will not exceed the polling period. Real-world applications often use a technique called *long polling*, where a client opens a connection and is willing to wait an arbitrary period of time for the server's response. The server can hold all of these long-poll connections open until there is a new event to distribute. The mechanics are standardized in recent browsers with the WebSockets protocol, providing an abstraction of bidirectional streams between clients and servers. Ur/Web presents an alternative abstraction (implemented with long polling) where servers are able to send typed *messages* directly to clients.

The messaging abstraction is influenced by concurrent programming languages like Erlang [1] and Concurrent ML [31]. 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**. Atomicity will be important in our chat-room example, to preserve a consistent global view of message order, when many simultaneous RPCs may be notifying the same set of clients with different messages.

The API for channels is straightforward:

$$\begin{array}{rcl} \mathsf{channel} & :: & \mathsf{Type} \to \mathsf{Type} \\ \mathsf{channel} & : & \forall\alpha.\ \mathsf{transaction}\ (\mathsf{channel}\ \alpha) \\ \mathsf{recv} & : & \forall\alpha.\ \mathsf{channel}\ \alpha \to \mathsf{transaction}\ \alpha \\ \mathsf{send} & : & \forall\alpha.\ \mathsf{channel}\ \alpha \to \alpha \to \mathsf{transaction}\ \{\} \end{array}$$

One wrinkle is that channel and send may only be called in server-side code, while recv may only be called in client-side code. A compiler analysis makes sure that functions are not called in the wrong places, and we have considered changing the types of the basic operations in a future Ur/Web version, so that standard type inference verifies sidedness properties.

Figure 6 gives another reimplementation 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`.

`queryI1` 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 **i**mperative side effects (justifying the `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 6 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. For instance, the `Log` module could use channels privately, without disrupting the use of channels in `Room`. Manual long-polling approaches require explicit dispatching logic, to the different parts of an application waiting for server notifications, because in practice a client can only afford to have one long-polling connection active at once.

Figure 6 contains no explicit deallocation of clients that have stopped participating. The Ur/Web implementation detects client departure using a heartbeating mechanism. When a client departs, the runtime system *atomically deletes from the database all references to that client's channels*. In this respect, channels in the database act like weak pointers. A database row with a field typed like `channel T` will be deleted outright when the owning client departs. When a row contains a field of type `option (channel T)`, which may hold either a channel or a null value, references to dead channels are merely nulled out. All clean-up operations described in the last two sentences commute with each other, so there is no need to specify the order in which different channel references for a single departed client are dealt with. Automatic deletion of channel references provides a kind of modularity similar to what garbage collection provides for heap-allocated objects.

Before wrapping up the tutorial part of the paper, we want to emphasize how this sequence of examples has demonstrated key Ur/Web design principles. Early in the sequence, we settled on one interface for the `Room` module. As we progressed through fancier implementations, no changes were required to the `main` function or any global application configuration file. Strong **encapsulation** of module state guarantees that code outside the module cannot distinguish between alternatives. Throughout all versions of the application, we also maintained the **simple concurrency** abstraction of one thread running at a time, with context switches only at well-defined points.

3. Implementation

The Ur/Web compiler is implemented in Standard ML, in about 55,000 lines of code. It compiles Ur/Web source files to run on the server, as native POSIX Threads binaries via C; and on the client, as JavaScript. Compiled applications are linked with a runtime system, with both a server part, in about 5,000 lines of C; and a client part, in about 1,500 lines of JavaScript.

We summarize here some of the less obvious aspects of the implementation.

3.1 Atomic Execution of Database Operations

The focus on atomic execution in Ur/Web is inspired by the database community's decades of experience with *transactions* [17], an idea that originated in that world. Many production-quality database servers support various flavors of atomicity. Since version 9.1, PostgreSQL has supported truly atomic (serializable) transactions [30]. In other words, it looks to arbitrary database clients as if their sequences of SQL operations execute without interference by other threads. We recommend using Ur/Web with PostgreSQL, but it is also possible to use other backends that do not quite provide serializable transactions, at the cost of weakening Ur/Web's operational semantics.

We have been surprised at the near absence of programming frameworks that take advantage of serializable or almost-serializable database transactions. Typically, database engines implement these abstractions with combinations of locking and optimistic concurrency. In either case, it is possible for operations to encounter serialization failures (with locking because of deadlock, and with optimistic concurrency because a conflict is detected). Popular SQL client libraries indicate serialization failure as just another error code returnable by any operation. Applications must check for this code after every operation, being prepared to do something sensible when it appears.

In contrast, Ur/Web hides serialization failures completely. Any serialization failure in an Ur/Web transaction execution *undoes and restarts the transaction*. The `transaction` monad is carefully designed so that all side effects may be undone as necessary. The Ur/Web runtime system uses C's `setjmp`/`longjmp` to allow reverting to the start of a transaction, from arbitrary code running within it. Since server threads share state only through the database, it is easy to run each remote call with a fresh private heap, so that undoing heap effects is a constant-time operation.

By hiding serialization failures automatically, Ur/Web is able to present to programmers the clean transactional abstraction sketched in the last section.

3.2 Message Routing with Channels

It would violate the transactional abstraction to send messages to clients immediately during `send` operations. Instead, `send`s are appended to a thread-local log. At a transaction's *commit* point, where all operations have completed without serialization failures, we atomically execute all message sends.

The server may not connect directly to clients to hand off messages. Instead, clients periodically open long-polling connections to the server. If a client is connected at the point where the server wants to hand off a message, the message is written directly to the client, and the connection is closed. Otherwise, messages for clients are appended to server-side buffers maintained per client. If a client connects again when its buffer is nonempty, all queued messages are sent immediately as the HTTP response, and the connection is closed.

The last (implicit) ingredient of message routing is garbage-collecting dead channels. A heartbeat mechanism detects dead clients. For each application, the compiler generates a C function parameterized on a client ID, to remove all database references to that client. Whenever a client departs, that C function is run inside a transaction of its own. Since client data structures may be reused across departures, this use of database transactions interacts critically with transactions for normal remote calls, to ensure that messages are not accidentally sent to new clients who have taken over the data structures of intended recipients who have already departed. From the perspective of an atomic remote call, any client is either fully present or fully absent, for the whole transaction.

3.3 Implementing Functional-Reactive GUIs

In earlier sections, we discussed how Ur/Web HTML fragments are implicitly instrumented for incremental recomputation via the `signal` monad. Each fragment is conceptually a function over the current values of all data sources, and the runtime system should understand well enough *which* sources matter for each fragment, to avoid wasteful recomputation of fragments that do not depend on a set of source changes. Implementing this semantics requires using custom data structures that are periodically transformed and installed into browsers' normal data structures for dynamic HTML trees, called the Document Object Model (DOM).

One key data structure appears in the JavaScript parts of the runtime system. Rather than representing HTML fragments as strings, we instead use what is morally an algebraic datatype with 3 constructors. The simplest constructor injects literal HTML strings. A two-argument constructor concatenates two values. Finally, one constructor injects a JavaScript function (compiled from Ur/Web code) that will be used as an event handler or signal, e.g. in an `onclick` or `<dyn> signal` attribute. This datatype is easily forced into normal HTML code, but the explicit closures within it force the JavaScript garbage collector to keep alive any objects that the handlers depend on.

A crucial piece of logic appears in the implementation of the `<dyn>` pseudotag, which injects signals into HTML fragments. The runtime system begins by evaluating the signal with the current data source values, producing a value in the datatype from the prior paragraph. That value is then forced into a normal HTML string, replacing all closure references with reads from positions of a global array where we have stashed the closures. In the new DOM tree node created from this code, we also record which slots in that closure array we have used. Additionally, the signal evaluation process saved the set of sources that were read. We only need to rerun the signal if any of these sources change. This set of sources is also saved in the new DOM node.

Data sources are represented as records containing data values and *sets of DOM nodes that are listening for changes*. Whenever a source changes, all listening nodes must be recomputed. Recomputation needs to deallocate the old HTML contents of a node, including freeing its global closures and removing it and its subtrees from the listener sets of sources. It is precisely to enable this cleanup that we save sets of closure slots and sources in each DOM node. Recomputation must also update these sets appropriately.

4. Evaluation

In this section, we evaluate the effectiveness of the Ur/Web implementation as a practical programming tool, considering both performance and programmer productivity. We start with a discussion of some microbenchmarks and then turn to a summary of deployed Ur/Web applications.

4.1 Microbenchmarks

Readers might worry that Ur/Web's higher abstraction level vs. mainstream frameworks may bring an unacceptable performance cost. We briefly consider performance via microbenchmarks, where in important cases we find exactly the opposite comparison to less abstract frameworks.

An Ur/Web solution was entered into the TechEmpower Web Framework Benchmarks[3], a third-party comparison initiative run independently of the author of this paper, using contributed benchmark solutions from different communities, including an Ur/Web solution written in part by the author. Evaluation centers on *throughput* of Web applications, or how many concurrent requests

[3] `http://www.techempower.com/benchmarks/`

their server-side pieces can handle per second. *Latency* (time to service a single request) is also measured, but not highlighted as much on the results pages.

Ur/Web does particularly poorly on a test with many database writes. Most frameworks in the competition do not enforce transactional semantics as Ur/Web does, and on every remote call we pay a performance cost for runtime analysis of transactional operations, paying the even higher cost of transaction restart when there is much contention. We hope to convince the benchmark coordinators to add a version of this test where implementations are required to use transactions.

However, on more realistic workloads, Ur/Web does very well. Exactly one test (`fortunes`) involves actual generation of HTML pages, based on results of database queries. On the benchmark's highest-capacity server, with 40 hardware threads all running Ur/Web server threads, Ur/Web achieves the best latency (2.1 ms) and the 4th-best throughput (about 110,000 requests per second) among all frameworks. This test (in Round 9 of the benchmarks) involved about 70 different configurations of about 60 different frameworks, including almost all of the most popular frameworks for real-world applications. We think these results are particularly encouraging given that our competitors are mostly not running within SQL transactions (i.e., they provide weaker semantics that make programming more challenging).

4.2 Deployed Applications

The Ur/Web implementation has been available as open source for long enough to develop a modest community of users. Beside the author of this paper, 12 users have contributed patches. Several serious Ur/Web applications have also been deployed on the Web. In this subsection, we summarize the ones we are aware of. Each is written by programmers outside the core Ur/Web development team, and there are no duplicate authors among the applications we survey, with one exception as noted below. Table 1 lists the applications and their URLs and authors. Nearly all of these applications also use Ur/Web's *foreign function interface (FFI)*, for interfacing with C and JavaScript libraries.

We especially want to highlight the first entry in the table, a profitable business based on an Ur/Web application. **BazQux Reader** is a Web-based reader for feeds of syndicated content, via formats like RSS and Atom. It provides similar functionality to the late Google Reader, but with many extras, like reading comments on feeds in the same interface. BazQux mostly uses Ur/Web's reactive GUI features and RPCs, interfacing via our FFI to a Riak[4] database that is kept up-to-date by a separate Haskell feed fetcher. As of this writing, there are about 2500 active BazQux users with paid subscriptions. Average load on the application is about 10 requests/second, with busy-period peaks above 150 requests/second. The Ur/Web source code of BazQux Reader is available publicly on GitHub[5].

The next two applications were developed by a single author. Both relate to the cryptographic currency bitcoin. The **Bitparking Namecoin Exchange** previously allowed visitors to trade bitcoins to do the equivalent of registering Internet domain names, within an alternative decentralized protocol for name mapping. It was decommissioned recently after 2 years of operation. The developer notes that most similar services were plagued by generic Web security problems (e.g., cross-site request forgery) in that period, but the Ur/Web-based exchange maintained a spotless record. The **Bitcoin Merge Mining Pool** has been running since 2011, managing a confederation of bitcoin miners. At its peak in 2013, about 10% of global progress in bitcoin mining worked through this application. It serves about 400 requests per second on average. This family of applications has mostly stressed Ur/Web's SQL interface, RPC system, and foreign-function interface (to connect to implementations of nontrivial cryptographic protocols).

The **Big Map of Latin America** is an animated presentation of that region's history. The user moves a slider widget to step through time, which places various icons around a colorful map, indicating events in categories like economics and the environment. Clicking on an icon opens an overlay with an illustrated article going into more detail. There is also a less-polished administrative interface for editing the history database with conventional HTML forms.

Ecosrv serves firmware upgrades for CGNAT network hardware devices, which connect to the Ur/Web application via an HTTP-based API. A more traditional GUI also displays statistics on recent activity. Ur/Web source code is available on GitHub[6].

Logitext is a graphical interface to sequent-calculus proofs, intended for use in teaching logic to undergraduates. Proof problems appear embedded within tutorial documents. Each solution is a tree of deduction steps, where the user adds a new step by clicking on the right subformula. The trees appear rendered in classic inference-rule notation. The backend for executing proof steps is the Coq proof assistant, invoked via RPCs and the FFI with some Haskell code involved.

Most of these applications' developers have no special connection to the Ur/Web development team, so their decisions to adopt Ur/Web provide some validation of the language design. We asked developers what they saw as the biggest positive and negative points of Ur/Web. There was a clear consensus that the biggest plus was the simple model for coding client-side GUIs that interact easily with server-side code. The most mentioned pain point was the way that the Ur/Web compiler checks that server-side code does not call functions that only make sense in the client, or vice versa. In designing the language, we were worried that it would be too complex to extend the type system to track such constraints, tagging monadic computation types with sidedness information. However, the approach we adopted instead, with ad-hoc static analysis on whole programs at compile time, leads to error messages that confuse even experienced Ur/Web programmers. In general, improvements to error messages and performance of the compiler (which does specialized whole-program optimization) are high on our list of future priorities.

One of the main motivations for designing the Ur language [8] was supporting complex metaprogramming, with detailed compile-time checking of metaprograms inspired by dependent type theory. As far as we know, the production Ur/Web applications are currently making only modest use of metaprogramming, e.g. for small convenience operations related to polymorphic record and variant types. The HTML and SQL encodings make extensive use of type-system features that go beyond those of ML and Haskell, but there has not yet been much direct adoption of such features outside the Ur/Web standard library.

5. Related Work

MAWL [2] was an early domain-specific language for safe programming of form-based Web applications. The MAWL compiler did static verification of HTML validity and of compatibility between forms and handlers. MAWL included no features for building libraries of abstractions, as it tied together forms and handlers via fixed sets of template files.

Continuation-based Web application systems make a different set of trade-offs than Ur/Web does in supporting abstraction. The PLT Scheme Web Server [20] provides completely first-class sup-

[4] `http://basho.com/riak/`

[5] `http://github.com/bazqux/bazqux-urweb`

[6] `http://github.com/grwlf/urweb-econat-srv`

Application	URL	Author
BazQux Reader	http://www.bazqux.com/	Vladimir Shabanov
Big Map of Latin America	http://map.historyisaweapon.com/	Daniel Patterson
Bitcoin Merge Mining Pool	http://mmpool.org/	Chris Double
Bitparking Namecoin Exchange [now defunct]	http://exchange.bitparking.com/	Chris Double
Ecosrv	http://ecosrv.hit.msk.ru/	Sergey Mironov
Logitext	http://logitext.mit.edu/	Edward Z. Yang

Table 1. Deployed Ur/Web applications

port for continuations-as-URLs, making it very easy to construct many useful abstractions. This platform works without a static type system, fitting the preferences of many programmers today. As a consequence, however, many kinds of strong encapsulation are impossible; for instance, the ability of programs to generate arbitrary HTML and JavaScript as strings means that any client-side component discipline can be subverted. Seaside [15] provides strong encapsulation guarantees at the expense of not exposing continuations as first-class URLs. Seaside's component system also only runs server-side; client-side code uses the normal JavaScript model, with the document exposed as a mutable global variable, with subtrees assigned names from a global namespace.

As far as we are aware, no continuation-based systems give persistent resources like database tables first-class status, which prevents encapsulation of these resources. That design decision makes it easier to build on top of existing programming languages and provide simpler libraries. We expect that different developers will prefer different positions in this design space. We hope that many fans of strong encapsulation in object-oriented and functional languages will prefer the style advocated in this paper. We have not found any previous recommendations of that style, so part of our mission in this paper is to add such a recommendation to the larger debate.

Mutable state enables new modularity disciplines. Web cells [26] are a variant of mutable references (implemented in the PLT Web Server) that provide the abstraction of each Web page having its own copy of the heap, while admitting efficient implementation. Initial Web cells implementations only worked with server-side storage of all continuations, which imposes significant storage requirements. More recent work [25] has shown how to represent Web-cell-using continuations compactly enough that these continuations may be serialized in full within URLs. Ur/Web's more static approach makes it possible for an algorithm to find all possible entry points to an application, enabling rigorous verification and testing as in some past work doing static analysis on Ur/Web code for security [9], while the PLT approach is more lightweight and easier for mainstream programmers to learn.

The Links [11] language pioneered strong static checking of multiple-tier Web applications, where the code for all tiers is collected in a single language. Where Ur/Web includes explicit markers (e.g., `rpc`) of control transfer between client and server, Links follows a more implicit approach [13], where different first-class functions are tagged as belonging to different sides and are automatically run on those sides when called. Links also includes a novel means of abstraction for HTML forms based on idioms [12]. Still, much of Ur/Web is inspired closely by Links and can be thought of as layering modularity features upon that foundation. Links does not include a module system or other vehicle for type abstraction, and any piece of Links code may access any database table or DOM subtree by referring to its textual name. Some other similar systems have been presented, including JWIG [10], which is a Java extension based around new types and program analysis.

Hop [33] is another unified Web programming language, this time dynamically typed and based on Scheme. Many elements are quite similar to the patterns demonstrated in our Ur/Web examples, including a simple RPC syntax (to server endpoints called *services*) and a quotation and antiquotation convention for easy mixing of server- and client-side code. GUI interaction follows the usual browser model, where a page is a mutable tree that can be walked arbitrarily by any client-side code. Hop is based on no distinguished database integration, instead supporting access to a variety of database systems via libraries. Hop servers support a novel means of configurable concurrency for pipeline execution [32], but we do not believe that any transaction-based model has been implemented, and it does not seem obvious how to implement such a configuration without more fundamental language support. The HipHop extension [5] uses synchronous reactive programming to support a different declarative style of organizing interactions between stages of processes that span clients and servers.

Ocsigen [3, 4] 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 concurrency model [36] is cooperative multithreading, on both client and server. The Ocsigen ecosystem includes libraries for both statically typed SQL database access and server-to-client message passing, the latter via a mechanism called *buses*. There is no support for grouping the two sorts of actions into transactions; rather, the semantics is the usual interleaving one, with cooperative context-switch points. The SQL library also allows arbitrary access to any table from any module that knows its string name, even allowing different modules to declare different versions of a table with different, incompatible types. There is optional library support for a functional-reactive GUI programming style.

The Opa language[7] is another statically typed unified language for database-backed Web applications. Its database integration is with the nonrelational system MongoDB[8], which does not support transactions, so Opa inherits a semantics that exposes interleaving of concurrent requests. Opa supports a precursor to the sort of database table encapsulation that we have described, where modules may declare private components of the database state. However, the Opa mechanism forces programmers to assign components names in a global namespace. The compiler will prevent duplicate use of a name, which thwarts threats against encapsulation, but which also forces authors of different modules to coordinate on a naming scheme. Similar issues arise in Opa's use of a mostly standard model of client-side GUIs, where elements are assigned textual names in a global namespace, and where the visible page is changed by mutating elements directly.

Strong encapsulation has been supported through capability systems [22], including the Capsules system [21] for isolating Web servlets. Capsules is built on top of Java, so, compared to Ur/Web, it provides an easier migration path for mainstream programmers. However, Capsules does not support database access, and, among cookies and other persistent values that are supported, true encapsulation is not possible. The reason is that a Capsules application must

[7] http://opalang.org/

[8] http://www.mongodb.org/

contain trusted code to delegate persistent resources to components, which implies that a component must know which resources its subcomponents use. In Ur/Web, components use special declarations like `table` to, in effect, seed their initial capability sets in a static and modular way.

Some ostensibly capability-safe subsets of JavaScript have been proposed, including Caja[9]. Formal analysis using operational semantics [24] has uncovered holes in some of these languages and suggested practical formal conditions that guarantee absence of holes. That line of work is complementary to our approach from this paper, as it provides a simple foundation without suggesting which abstractions should be built on it. With Ur/Web's current implementation, we depend on the fact that all modules are compiled from type-checked Ur/Web source. Instead, we could target some common safe JavaScript subset, ensuring safe interoperation with components built directly in JavaScript or with other compilers.

Several other languages and frameworks support functional-reactive programming for client-side Web GUIs, including Flapjax [28], which is available in one flavor as a JavaScript library; and Elm [14], 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 2.2.1, 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[10], 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 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, a standard DOM-based API for walking document structure and making changes imperatively is provided, though it is not very idiomatic. Meteor's machinery for reactive page updating involves a more complex API than in Ur/Web. Its central concept is of *imperative* functions that need to be rerun when any of their dependencies change, where 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 7 lines of code near the start of Section 2.2.1, 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.

[9] `http://code.google.com/p/google-caja/`

[10] `http://www.meteor.com/`

Other popular JavaScript frameworks include Angular.js[11], Backbone[12], Ractive[13], and React[14]. 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 7-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.

Microsoft's TouchDevelop [6, 35] is another recent Web language design, intended to appeal to novices. TouchDevelop allows client code to manipulate distributed data structures directly, applying distributed-systems techniques automatically to enforce eventual consistency. Consequences of such changes flow automatically into affected parts of documents, in a partly functional-reactive style. Allowing direct manipulation of data structures by clients raises security concerns, which Ur/Web avoids by funneling all such manipulations through RPCs from client to server, running code in a trusted environment where appropriate checks may be applied, with the chance to use message-passing channels to notify clients of updates in a globally consistent order. In general, the TouchDevelop model seems superior for newcomers to Web programming, while Ur/Web offers better robustness in a few senses.

The transactional abstraction has also been exposed in high-level programming languages via the software transactional memory model [34], for instance in GHC Haskell [18]. Such systems provide a transactional interface over the classic model of mutable, linked objects in a garbage-collected heap. Ur/Web's `transaction` monad is actually much closer to the original idea of transactions from databases. The only mutable server-side state in an Ur/Web application is in the SQL database (where columns have only primitive types) and in message queues of channels, so there is no need to do, e.g., conflict analysis on traces of accesses to pointer-based data structures.

6. 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. First, while mainstream practice is drifting further away from the conceptual simplicity of transactions and atomic execution of units of concurrent work, we suggest holding on to simple concurrency models, unless forced away from them by performance concerns. Second, while language-enforced encapsulation is widely praised for conventional data structures, we suggest that it ought to be generalized to key pieces of Web applications, like database tables and HTML subtrees, importing familiar software-engineering benefits with new twists. Ur/Web is already being used in serious production applications, and we hope languages adopting our principles can continue to help programmers realize nontrivial functionality with less effort and more confidence in correctness.

[11] `https://angularjs.org/`

[12] `http://backbonejs.org/`

[13] `http://www.ractivejs.org/`

[14] `http://facebook.github.io/react/`

Acknowledgments

For their bug reports, feature requests, patches, and enthusiasm, we thank the early adopters of Ur/Web, including the application authors acknowledged earlier but also many others. We also thank Gergely Buday, Jason Gross, Arjun Guha, Barbara Liskov, Clément Pit–Claudel, and the anonymous referees for their helpful feedback on drafts of this paper. This work has been supported in part by National Science Foundation grant CCF-1217501.

References

[1] J. Armstrong. Erlang – a survey of the language and its industrial applications. In *Proc. INAP*, pages 16–18, 1996.

[2] D. Atkins, T. Ball, M. Benedikt, G. Bruns, K. Cox, P. Mataga, and K. Rehor. Experience with a domain specific language for form-based services. In *Proc. DSL*, 1997.

[3] V. Balat. Ocsigen: typing Web interaction with Objective Caml. In *Proc. ML Workshop*, 2006.

[4] V. Balat, J. Vouillon, and B. Yakobowski. Experience report: Ocsigen, a Web programming framework. In *Proc. ICFP*, pages 311–316. ACM, 2009.

[5] G. Berry and M. Serrano. Hop and HipHop: Multitier web orchestration. In *Proc. ICDCIT*, pages 1–13, 2014.

[6] S. Burckhardt, M. Fahndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato. It's alive! Continuous feedback in UI programming. In *Proc. PLDI*, pages 95–104. ACM, 2013.

[7] J. Cheney, S. Lindley, and P. Wadler. A practical theory of language-integrated query. In *Proc. ICFP*, pages 403–416. ACM, 2013.

[8] A. Chlipala. Ur: Statically-typed metaprogramming with type-level record computation. In *Proc. PLDI*, pages 122–133. ACM, 2010.

[9] A. Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *Proc. OSDI*, pages 105–118, 2010.

[10] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Extending Java for high-level Web service construction. *TOPLAS*, 25(6):814–875, November 2003.

[11] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *Proc. FMCO*, pages 266–296, 2006.

[12] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. The essence of form abstraction. In *Proc. APLAS*, pages 205–220. Springer-Verlag, 2008.

[13] E. E. Cooper and P. Wadler. The RPC Calculus. In *Proc. PPDP*, pages 231–242. ACM, 2009.

[14] E. Czaplicki and S. Chong. Asynchronous functional reactive programming for GUIs. In *Proc. PLDI*, pages 411–422. ACM, 2013.

[15] S. Ducasse, A. Lienhard, and L. Renggli. Seaside – a multiple control flow Web application framework. In *European Smalltalk User Group – Research Track*, 2004.

[16] A. Frisch. OCaml + XDuce. In *Proc. ICFP*, pages 192–200. ACM, 2006.

[17] J. Gray. The transaction concept: Virtues and limitations (invited paper). In *Proc. VLDB*, pages 144–154, 1981.

[18] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proc. PPoPP*, pages 48–60. ACM, 2005.

[19] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. In *Proc. ICFP*, pages 11–22. ACM, 2000.

[20] S. Krishnamurthi, P. W. Hopkins, J. McCarthy, P. T. Graunke, G. Pettyjohn, and M. Felleisen. Implementation and use of the PLT Scheme Web Server. *Higher Order Symbol. Comput.*, 20(4):431–460, 2007.

[21] A. Krishnamurthy, A. Mettler, and D. Wagner. Fine-grained privilege separation for Web applications. In *Proc. WWW*, 2010.

[22] H. M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984. ISBN 0932376223.

[23] D. MacQueen. Modules for Standard ML. In *Proc. LFP*, pages 198–207. ACM, 1984.

[24] S. Maffeis, J. Mitchell, and A. Taly. Object capabilities and isolation of untrusted Web applications. In *Proc. IEEE S&P*, pages 125–140, 2010.

[25] J. McCarthy. Automatically RESTful Web applications or, marking modular serializable continuations. In *Proc. ICFP*. ACM, 2009.

[26] J. McCarthy and S. Krishnamurthi. Interaction-safe state for the Web. In *Proc. Scheme and Functional Programming*, 2006.

[27] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling objects, relations and XML in the .NET framework. In *Proc. SIGMOD*, pages 706–706. ACM, 2006.

[28] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: A programming language for Ajax applications. In *Proc. OOPSLA*, pages 1–20. ACM, 2009.

[29] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Proc. POPL*, pages 71–84. ACM, 1993.

[30] D. R. K. Ports and K. Grittner. Serializable Snapshot Isolation in PostgreSQL. *Proc. VLDB Endow.*, 5(12):1850–1861, Aug. 2012.

[31] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999. ISBN 0-521-48089-2.

[32] M. Serrano. Hop, a fast server for the diffuse web. In *Proc. COORDINATION*, pages 1–26. Springer-Verlag, 2009.

[33] M. Serrano, E. Gallesio, and F. Loitsch. Hop, a language for programming the Web 2.0. In *Proc. DLS*, 2006.

[34] N. Shavit and D. Touitou. Software transactional memory. In *Proc. PODC*, pages 204–213. ACM, 1995.

[35] N. Tillmann, M. Moskal, J. de Halleux, and M. Fahndrich. TouchDevelop: Programming cloud-connected mobile devices via touchscreen. In *Proc. ONWARD*, pages 49–60. ACM, 2011.

[36] J. Vouillon. Lwt: A cooperative thread library. In *Proc. ML Workshop*, pages 3–12. ACM, 2008.

[37] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. POPL*, pages 60–76. ACM, 1989.