1. Warm ups
   1.1. Domain
      1.1.1. Web programming: Single Tier Client Server
   1.2. Design Goals
      1.2.1. Encapsulation
      1.2.2. Simple Concurrency
      1.2.3. One program for clients and server
      1.2.4. Strong typing: Safety
         1.2.4.1. from code injection
         1.2.4.2. from validation errors
      1.2.5. Easily embedded and extensible
      1.2.6. close integration with SQL (Transactional State)
      1.2.7. functional reactive programming
      1.2.8. Security?
   1.3. Architecture and components
      1.3.1. Server
         1.3.1.1. Backend language (Anything)
      1.3.2. Client
         1.3.2.1. Javascript
         1.3.2.2. CSS
         1.3.2.3. HTML
         1.3.2.4. Java Applets
         1.3.2.5. Macromedia Flash!!!!!!111
      1.3.3. Database
         1.3.3.1. SQL
   1.4. Disadvantages of the way typical modern web apps are built
      1.4.1. 2 or 3 tiered architecture makes it hard to reason
      1.4.2. Programmer requires diverse skills and knowledge of many things
         1.4.2.1. Corollary: Programmer usually does not posses these skills, and the result is a bloated disaster.
      1.4.3. Up to programmer to enforce structure
         1.4.3.1. MVC architecture isn’t enforced in these technologies / languages
         1.4.3.2. Objects must be converted between types in different languages.
      1.4.4. Human sacrifice is required to make a modern web app work correctly and in a performant manner.

2. Design Evaluation
   2.1. What programming model does Ur/Web provide?
      2.1.1. Unified programming model that hides most web protocols, and only exposes HTML (for page structure) and SQL (for persistent data access)
      2.1.2. Enforces what we would typically call MVC structure
         2.1.2.1. Model -> embedded persistent SQL
         2.1.2.2. View -> client FRP GUI scripting
2.1.2.3. Controller -> links call out to RPC server-side computation

2.2. What abstractions and features does Ur/Web provide

2.2.1. Native SQL tables
2.2.2. RPCs
2.2.3. Strong type system
2.2.3.1. Regex types to strongly enforce XML systems
2.2.4. Channels and message passing
2.2.5. Encapsulation of modules
2.2.6. Links as functions, functions as human readable links

2.3. Explain the code in Figure 1

2.3.1. Define two SQL tables
2.3.1.1. store room and message data
2.3.2. Uses RPCs as links, defined as functions
2.3.3. Uses Embedded SQL queries
2.3.3.1. populate data fields that get put into the html template code

2.4. Figure 2

2.4.1. Define a Room module that can be reused, wrapping the code from Fig. 1
2.4.2. Fun Main uses MapX to generate templates using the Room module for each room id in the table

2.5. What are sources and how are they used

2.5.1. A source of data in a series of transactions that make up a stream
2.5.2. mutable reference wrapped in a monadic structure
2.5.3. Atomically safe
2.5.4. Different levels have same name, source as a value constructor and source as a type constructor

2.6. Explain Fig. 4

2.6.1. Callback functions written in antiquotes as a lambda function
2.6.1.1. Monadic structure
2.6.2. use source to handle the chat message stream
2.6.3. handles reactive gui programming on client side

2.7. Explain channels

2.7.1. RPC with bi-directional communication between client and server
2.7.2. Message abstractions
2.7.3. Guarantees messages get delivered to correct client
2.7.4. guarantees atomic calls
2.7.5. Implemented by long polling
2.7.6. Clients can be stored in the database and managed by GC

2.8. Figure 6

2.8.1. chat log is sourced on the listener callback which receives messages from the server over the channel
2.8.2. the channel is stored in the database
2.8.3. this means you can receive messages without having to say something yourself (the source is reactive to this server -> client call)

2.9. What features contribute to concurrency?
2.9.1. Mutable data is wrapped in monads -- everything else is immutable and therefore incorruptible
2.9.2. Atomic operations
2.9.3. Database and RPC
2.9.4. Coöperative multithreading model
2.9.5. Transactional state
2.9.6. Runtime handles channel communication

2.10. What features contribute to encapsulation?
2.10.1. Structure/Signatures
2.10.2. Module/interface structure whereby modules have private data (CSS, ajax handlers, etc.) that can only be accessed through those modules
2.10.3. SQL tables captured by the code and managed implicitly

2.11. What features contribute to security?
2.11.1. Modules restrict interfaces between elements
2.11.2. RPCs allow code to run on a trusted environment
2.11.3. Protection from SQL injection

2.12. Implementation
2.12.1. Mostly ML, but also JS and C
2.12.2. Built on top of Ur
2.12.3. Channels send heartbeats to detect dead clients
   2.12.3.1. If dead, remove all database references

2.13.
2.13.1. Evaluation
   2.13.1.1. Comparison with existing tools
   2.13.1.2. Practicality, real world uses
   2.13.1.3. Microbenchmarks by a 3rd party
      2.13.1.3.1. Performed badly on repeated writes because of transactions
      2.13.1.3.2. Performed well on HTML generation
   2.13.2. Improvements to evaluation
      2.13.2.1. Justification of sufficiency
      2.13.2.2. Approachability for non-web programmers
      2.13.2.3. Comparison on a benchmark that requires transactionality
      2.13.2.4. Evaluate security?

3. Evaluation as a DSL
3.1. What are the advantages and disadvantages of embedding Ur/Web in Ur?
3.1.1. Advantages
   3.1.1.1. Strongly typed
   3.1.1.2. Code must work
   3.1.1.3. Modules and encapsulation
3.1.2. Disadvantages
   3.1.2.1. Perhaps too strongly typed
   3.1.2.2. Nobody knows Ur
   3.1.2.3. Locked into SQL? (At least locked into SQL syntax)
3.2. Describe Ur/Web's runtime system. How does it contribute to the goals of the language?
   3.2.1. Client
       3.2.1.1. Runtime in JavaScript (1500 lines).
   3.2.2. Server
       3.2.2.1. Runtime is 5000 lines of C. Manages HTTP requests, and also SQL resources.
       3.2.2.2. SQL server

3.3. No, yes
3.4. No, Maybe?
3.5. Yes
3.6. ...
3.7. It does achieve the goals of the language but doesn’t appear to be poised for success
3.8. ...
3.9. The language itself contains great amounts of good and evil, and the paper can be confusing at times, though the evaluation stood out in a positive manner.