

UrWeb

Functional. Pure. Principled. Eager. Slightly mad.

Sean Chalmers

Sun 26 Apr 2015 14:18:00 AEST

Introduction

Brief introduction to the Ur language and its current incarnation in Ur/Web.

- Going Badless
- XHTML
- SQL
- JavaScript ... sort of
- Type System
- Limitations

Ur

Ur is a relative of Haskell and ML.

- Pure
- Statically typed
- Strict
- Type level programming
- Row types
- Type classes too!

Ur/Web

- **Ur** is currently unavailable outside of **Ur/Web**.
- **Ur/Web** is **Ur** plus a special standard library, plus a special compiler, purpose built for SQL backed web apps.
- All designed so that well-typed **Ur/Web** programs don't 'go wrong'.

Woo!

Ur/Web applications do not...

- Suffer from any kinds of code-injection attacks
- Return invalid HTML
- Contain dead intra-application links
- Have mismatches between HTML forms and the fields expected by their handlers
- Include client-side code that makes incorrect assumptions about the "AJAX"-style services that the remote web server provides
- Attempt invalid SQL queries
- Use improper marshaling or unmarshaling in communication with SQL databases or between browsers and web servers

Going Badless
Badless Double Down
Bits
Type System

XHTML / HTML
Query-Tastic
Query-Tastic
No More JavaScript
More Fun
More?!

XML Built In

- XHTML (and HTML5) are built into the language.
- Structure is also **verified** by the language.

XML Built In (yay?)

Compile time checks of structure:

```
<xml><body>  
  <h1>Woot</h2> <-- compile error  
</body></xml>
```

Prevents noobing it up from simple errors:

```
<xml><body>  
  <form action={nomForm}>  
<form> <input type="text"> </form> <-- illegal subform  
  </form>  
</body></xml>
```

SQL Built In !

- Supports Postgresql as default, as well as MySQL, and SQLite.
- SQL queries are made up of functions as part of the base library.
- Types for queries and tables are checked at compile time.
- Database creation SQL is provided by the compiler.

SQL Built In !

Make a table:

```
table foo : { Id : int, Buzz : string, Created : time }  
  PRIMARY KEY Id
```

Query, just a little bit:

```
fun list () =  
  rows <- queryX (SELECT * FROM foo) (fun row =>  
    <xml><div><h1>{[row.Foo.Buzz]}</h1>  
    <p>Created: {[row.Foo.Created]}</p>  
    </div></xml>  
  );  
return  
  <xml>{rows}</xml>
```

SQL Built In !

```
table a : { Id : int, Foo : string }
```

Compile error!

```
queryL1 (SELECT a.Foo FROM a WHERE a.Id = 'buzz')
```

Compile error!

```
queryL1 (SELECT a.Bar FROM a WHERE a.Id = 1)
```

Front End McGuffins

If you didn't have to write the JavaScript, is it really badless JavaScript?

- No difference between back or front end UrWeb code.
- Compiler works out which code is required where and compiles accordingly.
- Signal based Functional Reactive Programming.
- Communication by 'rpc' and typed 'channels'.

Front End McGuffins

Dynamic html is simple to include:

```
<dyn signal={s <- signal s; doSomethingAmazing s}/>
```

All your favourite events are there too:

```
<button onclick={fn evt => foo evt}/>
```

Other Bits

- Output is a single executable that can be the web or fastcgi server.
- The binary produced is extremely efficient compiled C code.
- The binary does not use garbage collection, relying instead on a technique known as 'region based memory management'.
- It is "stupid fast". Refer to the Techpower Benchmarks if you're into such things.

Bit more

All requests run inside a 'transaction', analogous to 'IO' from Haskell.

```
val readBack : transaction int =  
  src <- source 0;  
  set src 1;  
  n <- get src;  
  return (n + 1)
```

EVERYTHING is inside a transaction. Postgres is supported by default due to its strong support for transactions.

GIEF TYPES

- Type inference
- Parametric Polymorphism

```
fun id [a] (x : a) : a = x
```

Some higher order functions:

```
mp : a ::: Type -> b ::: Type -> (a -> b) -> t a -> t b
```

```
append : a ::: Type -> t a -> t a -> t a
```

Polymorphic datatypes:

```
datatype list a = Nil | Cons of a * list a
```

```
val emptyList = Nil
```

```
val notEmpty = Cons (3, (Cons (2, (Cons (1,Nil)))))
```


Anonymous Records

```
val a = { A = 42, B = "Fred" }  
  
type b = { A : int, B : string }  
  
a.A == 42
```

Typeclasses!

```
fun max [a] (_ : ord a) (x : a) (y : a) : a =  
  if x < y then  
    y  
  else  
    x
```

Limitations

- Currently, the type errors are terrifying.
- First Class / Impredicative polymorphism is powerful but complex.
- These aren't the functors you're looking for. `Option.mp` & `List.mp` *sadface*
- Ur/Web applications are not very social: making requests to external API requires use of the FFI (ps, it has a JavaScript and C FFI).
- Defining Typeclasses is not as straight forward as Haskell.