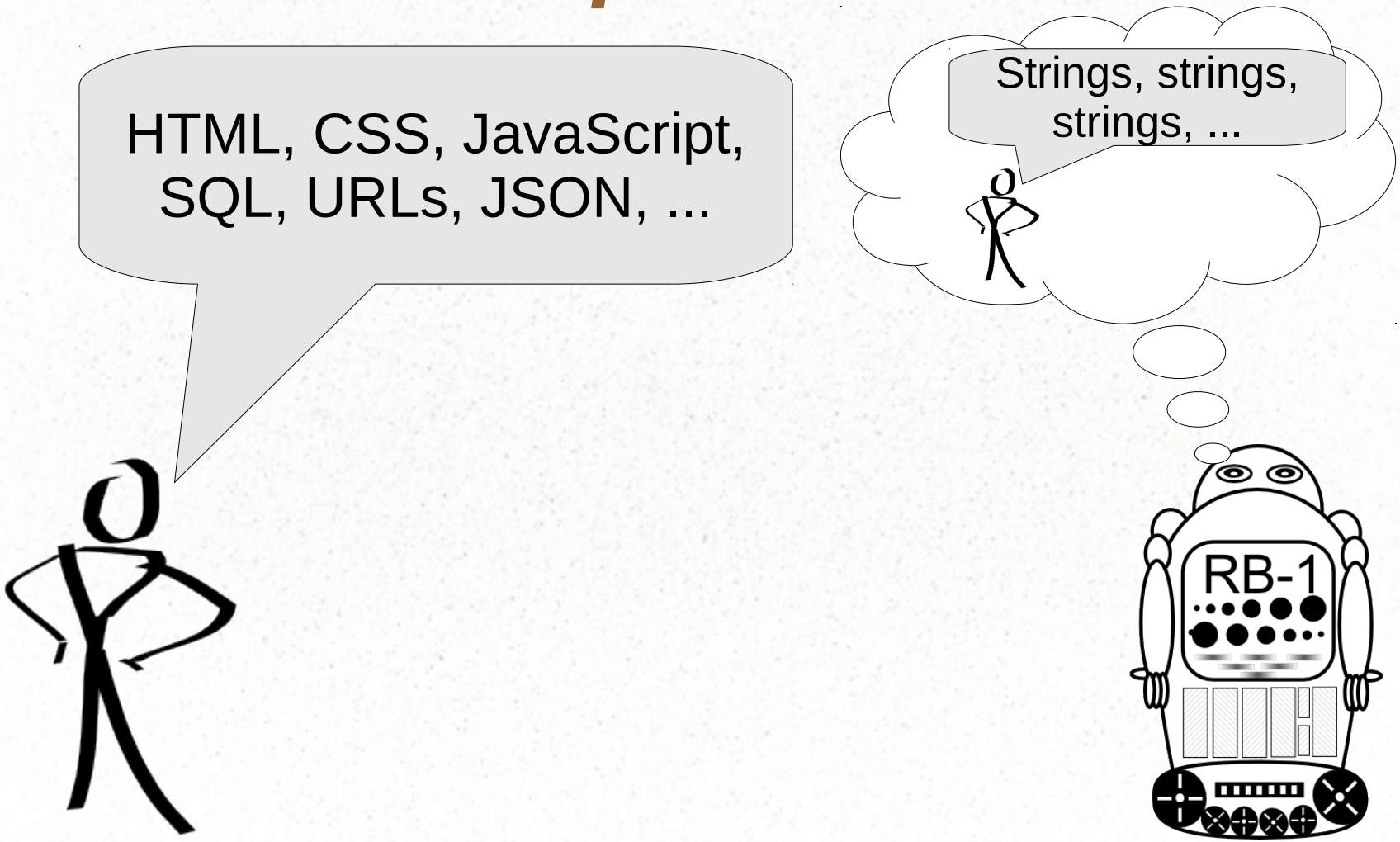
Ur/Web, a Domain-Specific Functional Programming Language for Modern Web Applications

Adam Chlipala

Web Applications: A Steaming Pile of Text



Saner Languages/Frameworks

- Links
- Ocsigen
- LINQ
- WebSharper
- OPA (“One-Pot Application”)
- ...?

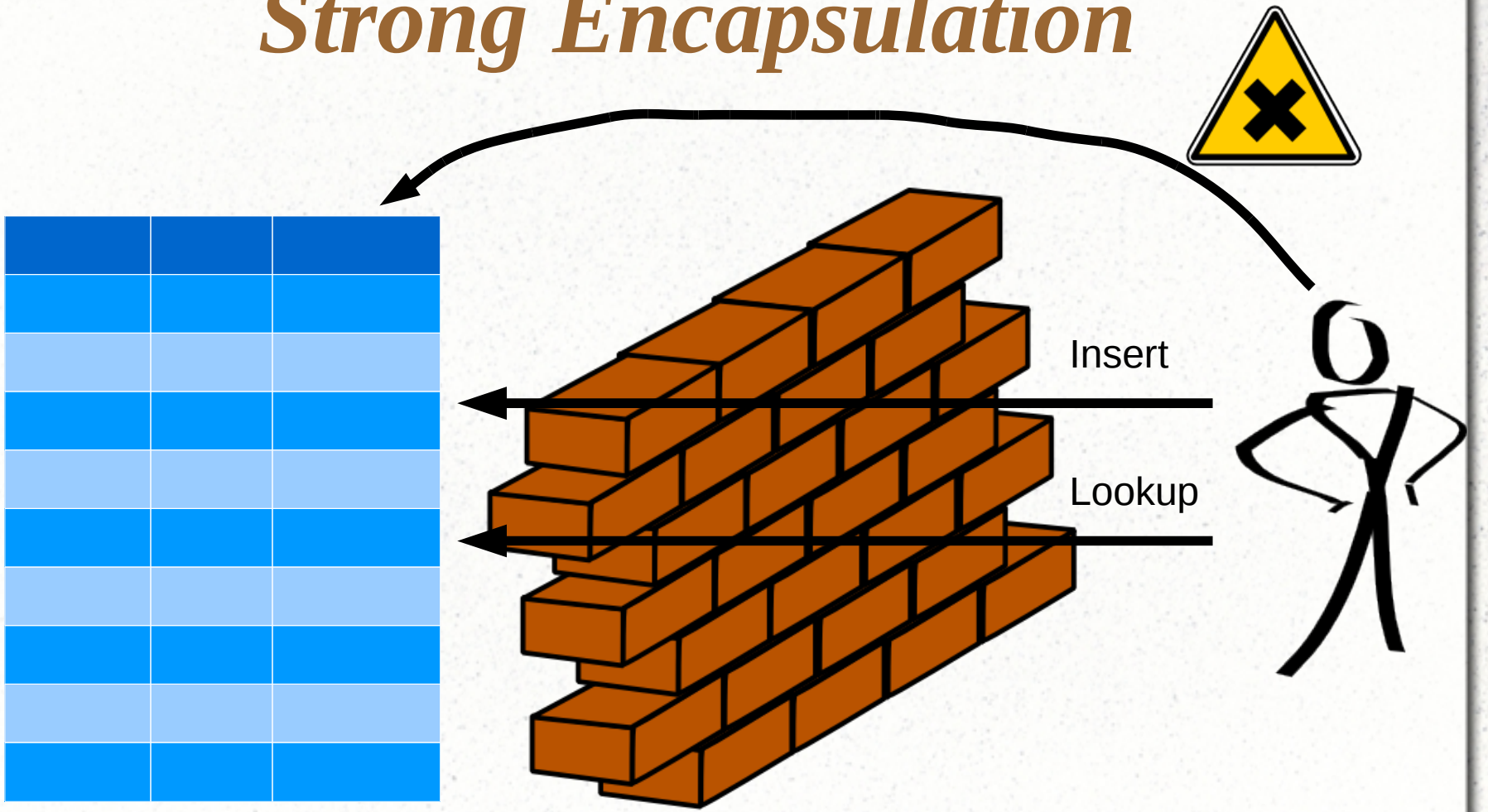
Executive Summary

Ur/Web is a **domain-specific language**
with a **fancy static type system**
with **first-class support** for Web app architecture
including **strong encapsulation**
and **statically-checked metaprogramming**
with **security by construction**

Hello World!

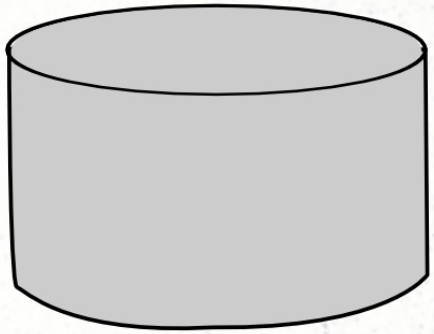
```
fun main () = return <xml>
  <head>
    <title>Hello world!</title>
  </head>

  <body>
    <h1>Hello world!</h1>
  </body>
</xml>
```

Strong Encapsulation

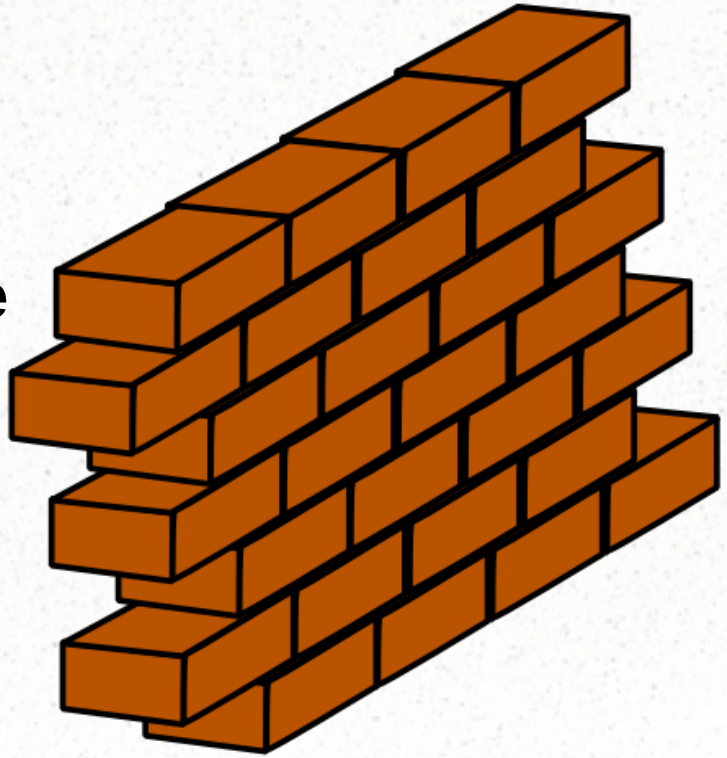


Hash Table **Module Interface** **Client Code**

Web 1.0 Encapsulation

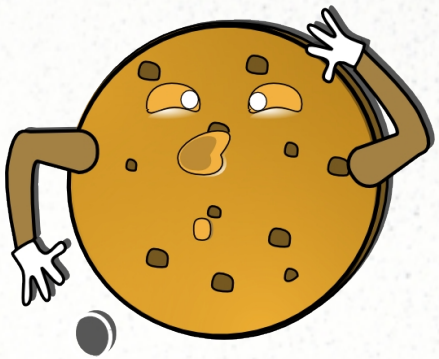


Users Database Table



Module Interface

Client Code



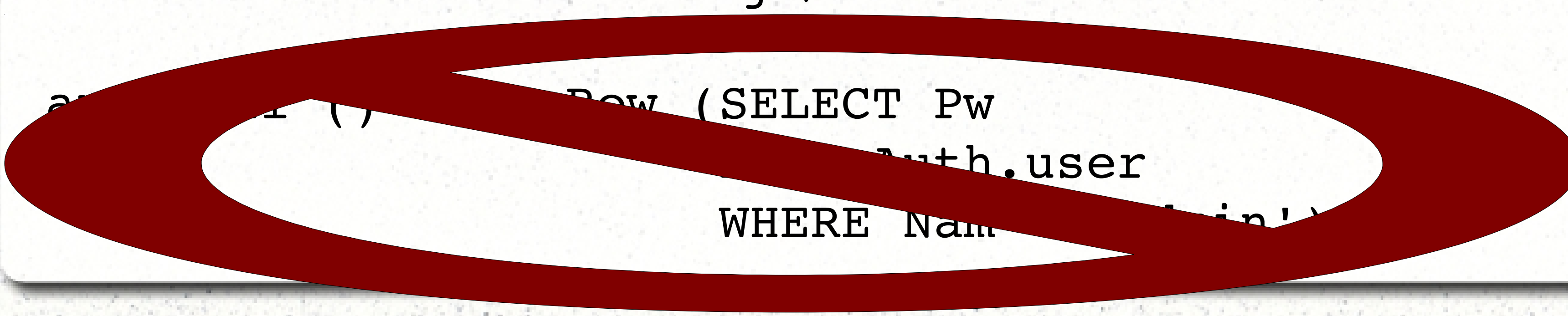
Login Cookie

Encapsulation

```
structure Auth
    : sig
        val loginForm : unit -> xbody
        val whoami : unit -> string
      end = struct
    table users : {Nam : string, Pw : string}
    cookie auth : {Nam : string, Pw : string}

    fun rightInfo r =
       oneRow (SELECT COUNT(*) FROM users
               WHERE Nam = {[r.Nam]}
                 AND Pw = {[r.Pw]}) = 1

    fun login r = if rightInfo r then setCookie auth r
                  else error "Wrong info!"

    fun loginForm () = (* Form handled by 'login'... *)

    fun whoami () = let r = getCookie auth in
                    if rightInfo r then r.Nam
                    else error "Wrong info!"
end
```

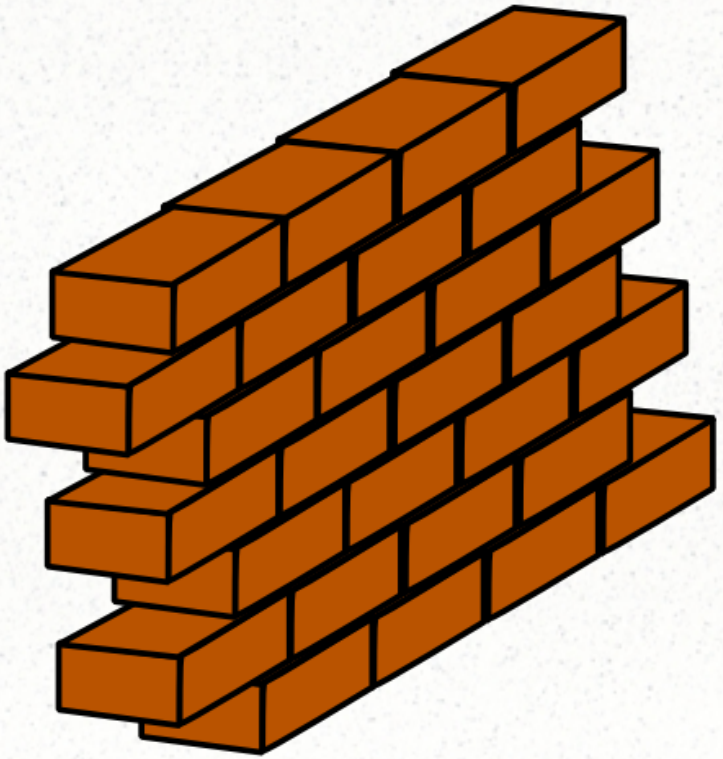
Some Client Code

```
fun main () = return <xml>
   {Auth.loginForm ()}

   <p>Welcome.  You could
   <a link={somewhere ()}>go somewhere</a>.</p>
</xml>

and somewhere () =
   user <- Auth.whoami ();
   if user = “Fred Flintstone” then
      return <xml>Yabba dabba doo!</xml>
   else
      return <xml>Boring</xml>
```



Web 2.0 Encapsulation

Subtree of Dynamic Page Structure

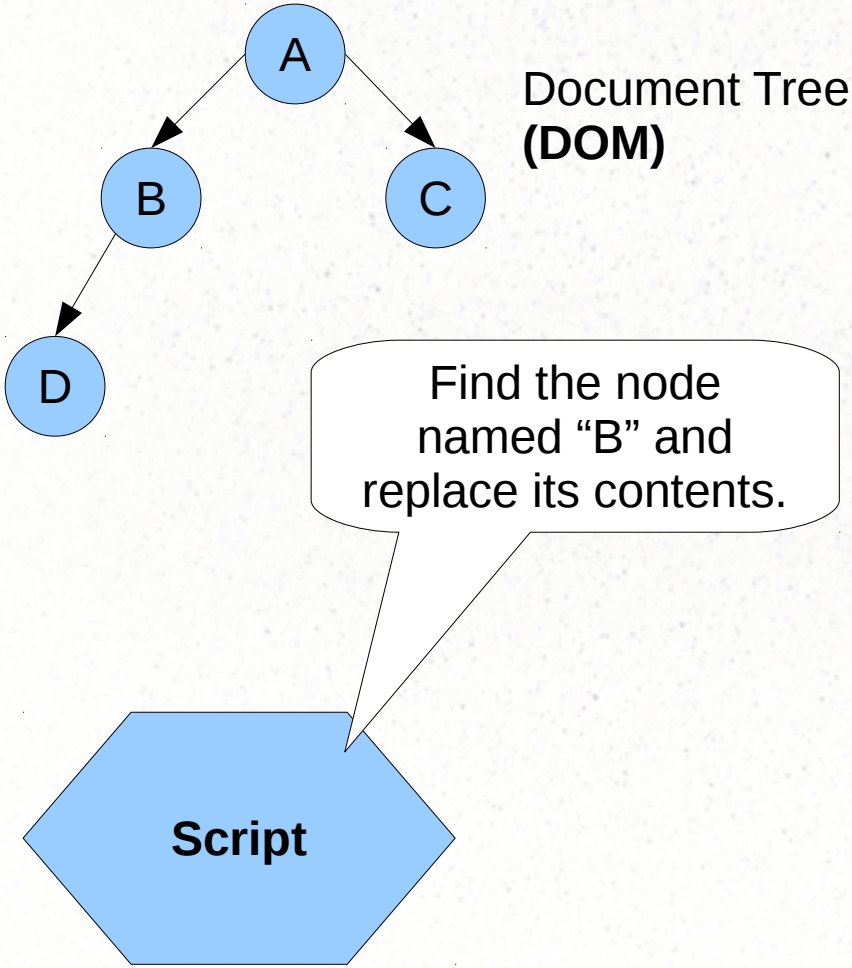


Module Interface

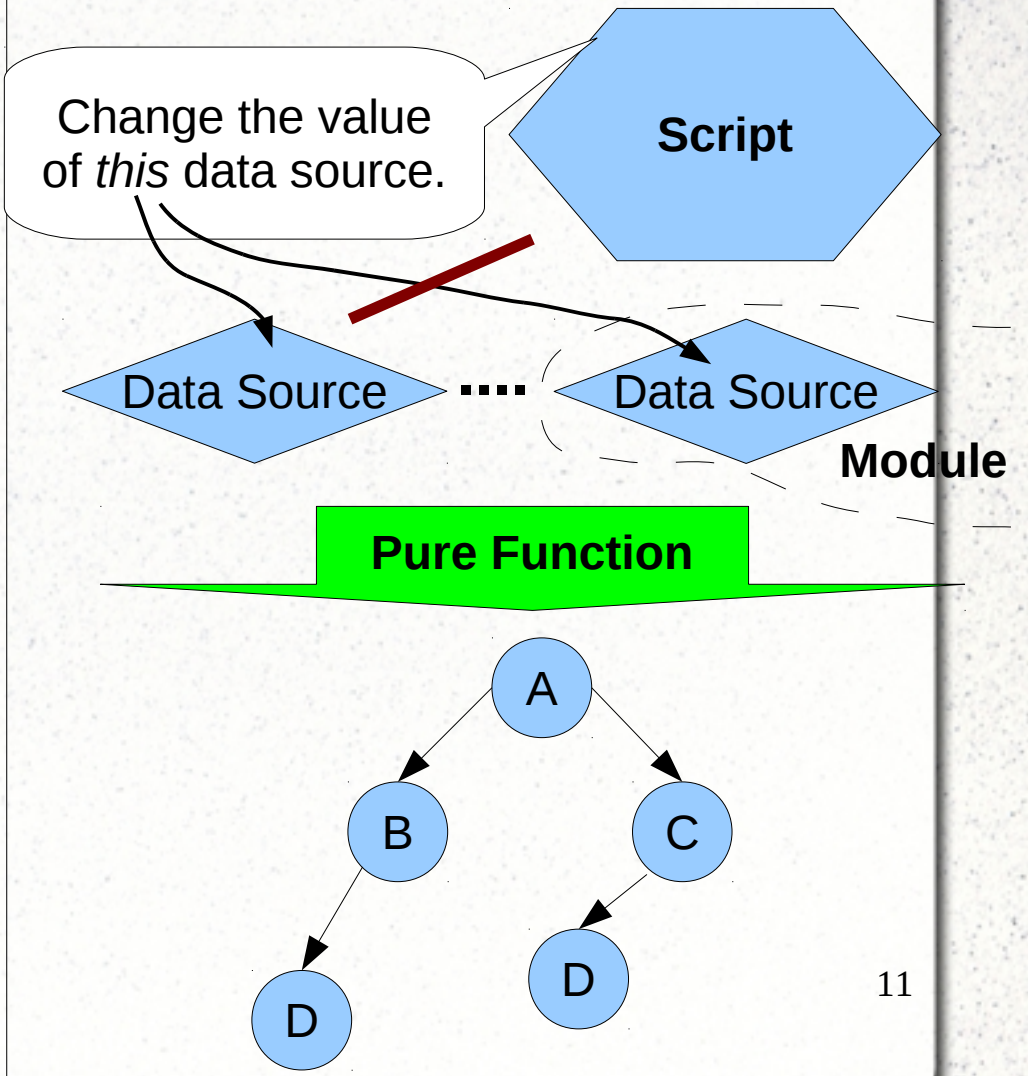
Client Code

Functional-Reactive GUIs

The Status Quo:



The Reactive Way:



A Client-Side Counter

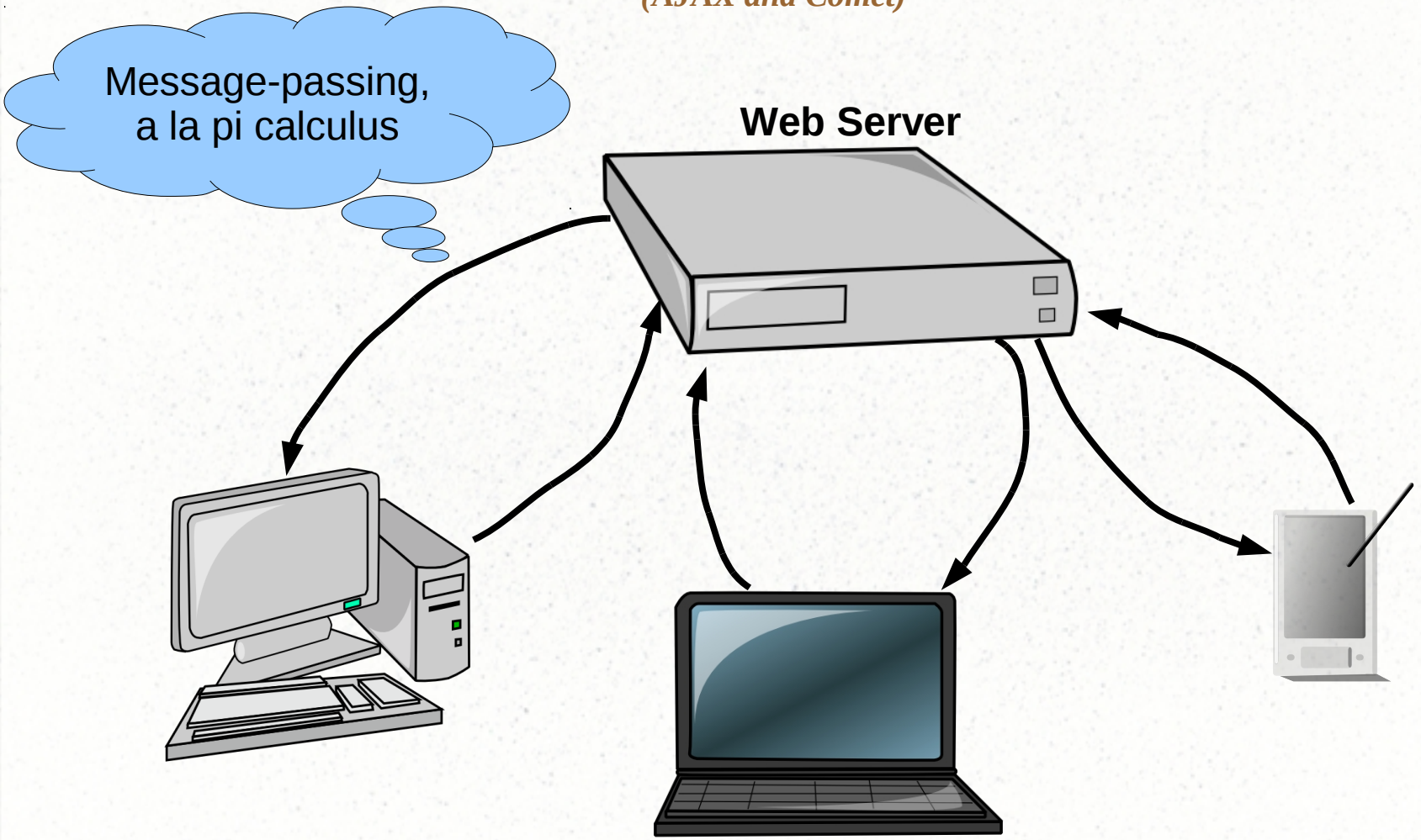
```
structure Counter : sig
                      type t
                      val new : unit -> t
                      val increment : t -> unit
                      val render : t -> xbody
                    end = struct
  type t = source int

  fun new () = source 0

  fun increment c = n <- get c; set c (n + 1)

  fun render c = <xml>
    <dyn signal={n <- signal c;
                 return <xml><b>{[n]}</b></xml>}/>
  </xml>
end
```

Counter Client

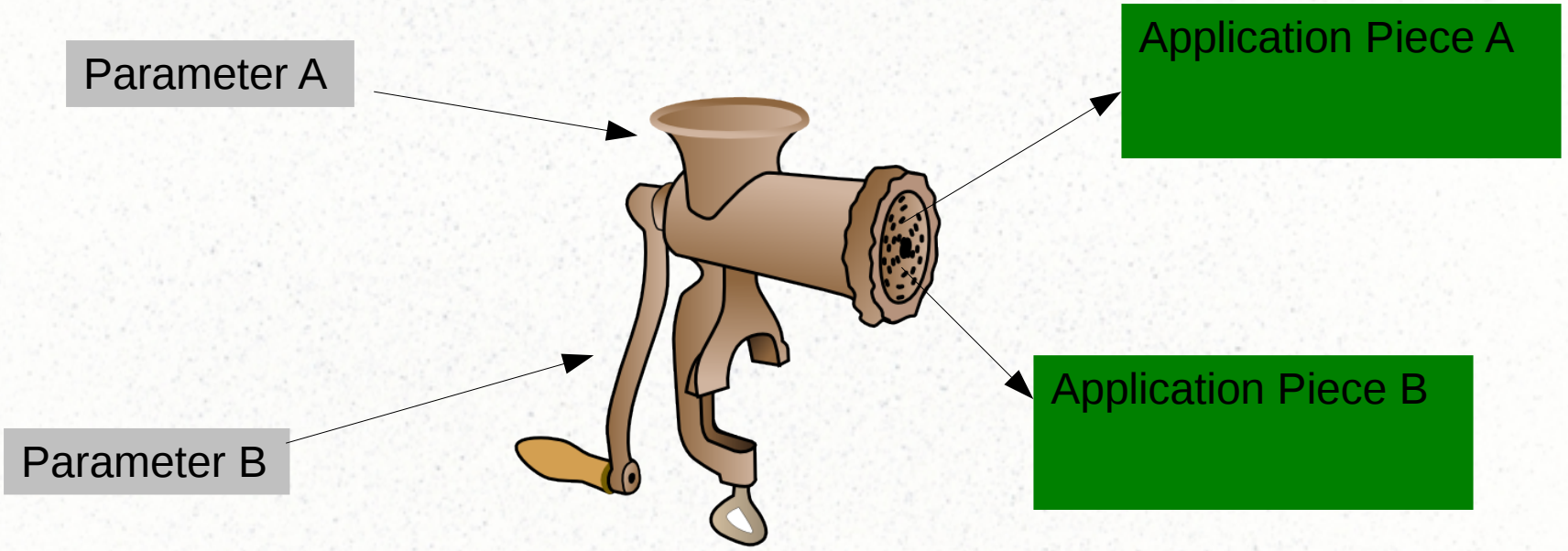
```
fun main () =
   c <- Counter.new ();
   return <xml>
      {Counter.render c}
      <button onclick={Counter.increment c}/>
      Backup copy: {Counter.render c}
      <button onclick={set c -1}/>
   </xml>
```

Client-Server Communication

(AJAX and Comet)



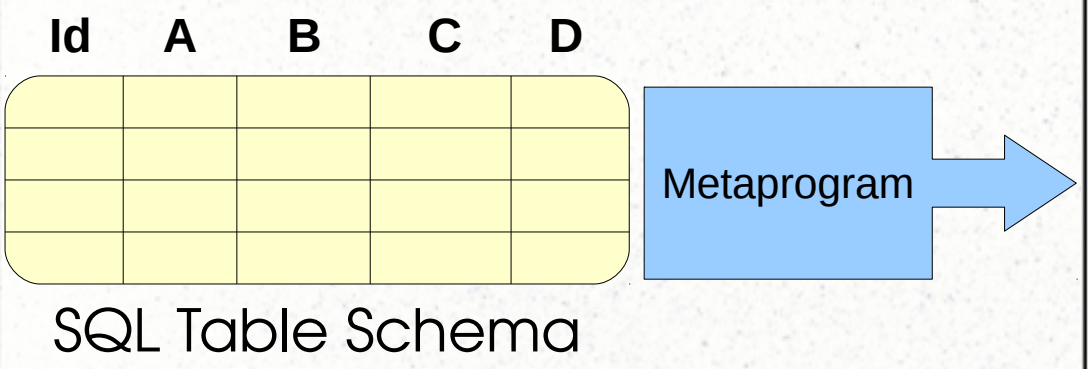
Metaprogramming for the Web



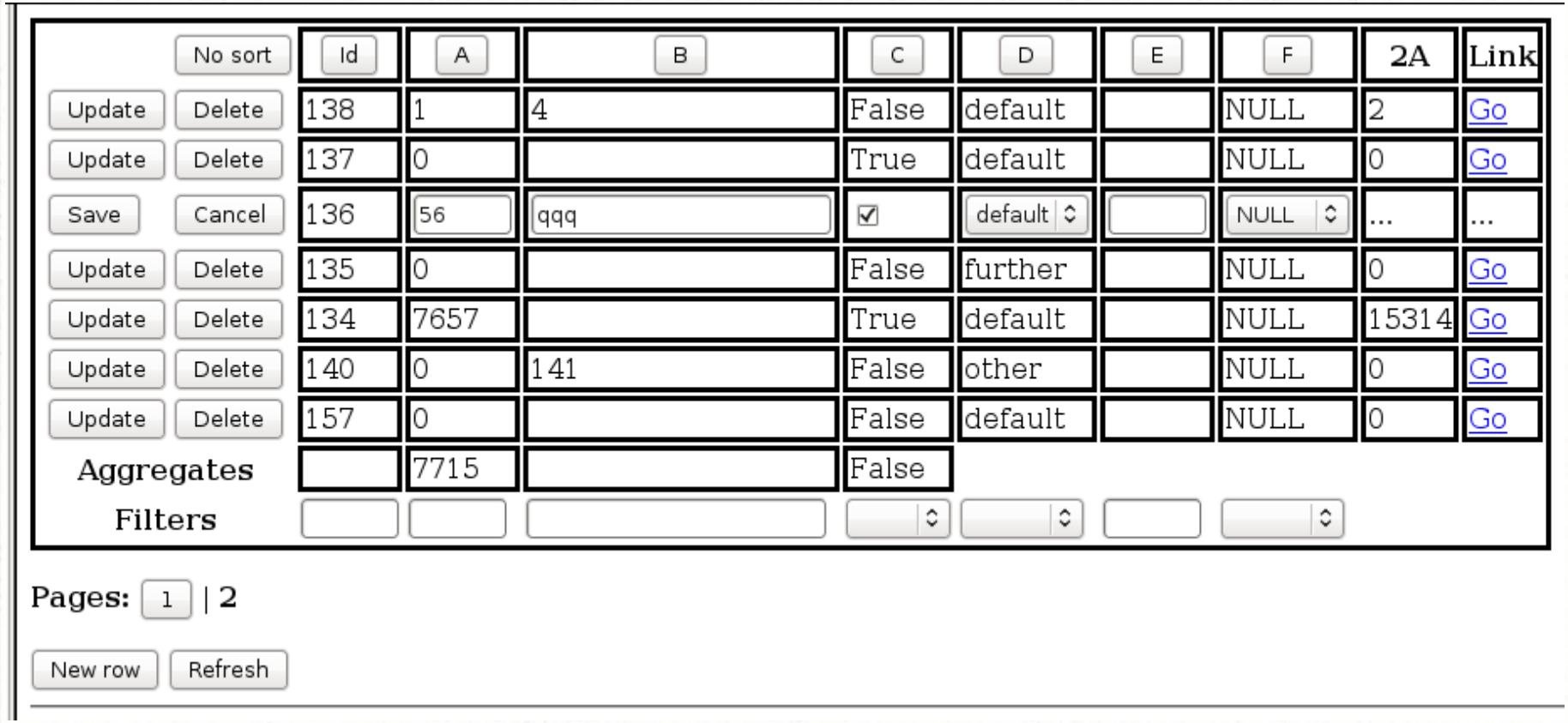
Parameterized code generator

Already very popular in Ruby on Rails and other frameworks!

Automatic Admin Interface



In-Browser Spreadsheet



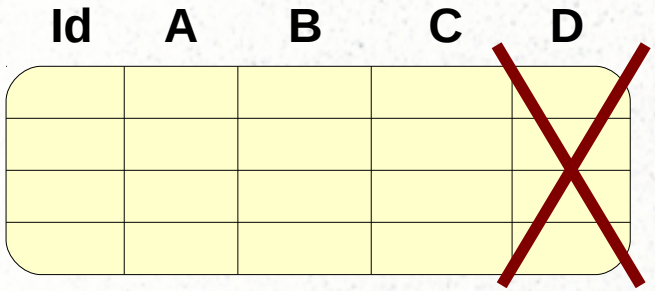
	No sort	Id	A	B	C	D	E	F	2A	Link
Update	Delete	138	1	4	False	default		NULL	2	Go
Update	Delete	137	0		True	default		NULL	0	Go
Save	Cancel	136	56	qqq	☑	default		NULL	...	...
Update	Delete	135	0		False	further		NULL	0	Go
Update	Delete	134	7657		True	default		NULL	15314	Go
Update	Delete	140	0	141	False	other		NULL	0	Go
Update	Delete	157	0		False	default		NULL	0	Go
Aggregates			7715		False					
Filters										

Pages: 1 | 2

New row | Refresh

Ad-Hoc Code Generation?

Edit some source files to customize....
Now change the database schema....



SQL Table Schema

Metaprogram

```
app
  models
    post.rb
  views
    posts
      index.html.erb
      show.html.erb
      ....
  config
    routes.rb
  ....
```

```
script/generate scaffold T
  Id:int A:string B:float
      C:string D:int
```

Metaprogramming in a Nutshell

Compile-time Programming Language
(AKA, type system)
(not Turing complete, but includes basic lambda calculus)
Crucial feature: **type-level records!**

Included in

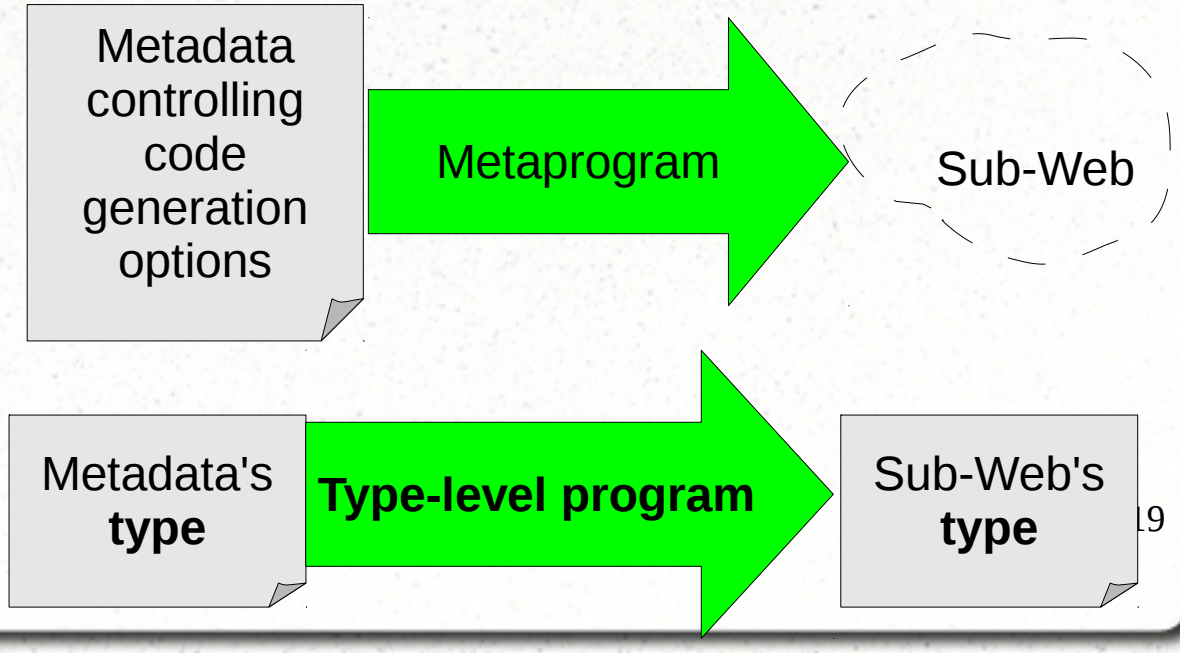
Run-time Programming Language
(Turing complete)

Key Property 1:

These types are expressive enough to guarantee absence of code injection, abstraction violation, etc..

Key Property 2:

Effective static checking that the metaprogram really has the claimed type



Core Language Features

κ ::= Type | $\kappa \to \kappa$

τ ::= $\tau \to \tau$ | α | $\forall \alpha :: \kappa.\ \tau$ | $\tau\ \tau$ | $\lambda \alpha :: \kappa.\ \tau$

e ::= x | e e | λx : τ. e | e [τ] | $\Lambda \alpha :: \kappa.$ e

Core Language Features

κ ::= Type | $\kappa \to \kappa$
| Name | $\{\kappa\}$
τ ::= $\tau \to \tau$ | α | $\forall \alpha :: \kappa. \tau$ | $\tau\ \tau$ | $\lambda \alpha :: \kappa. \tau$

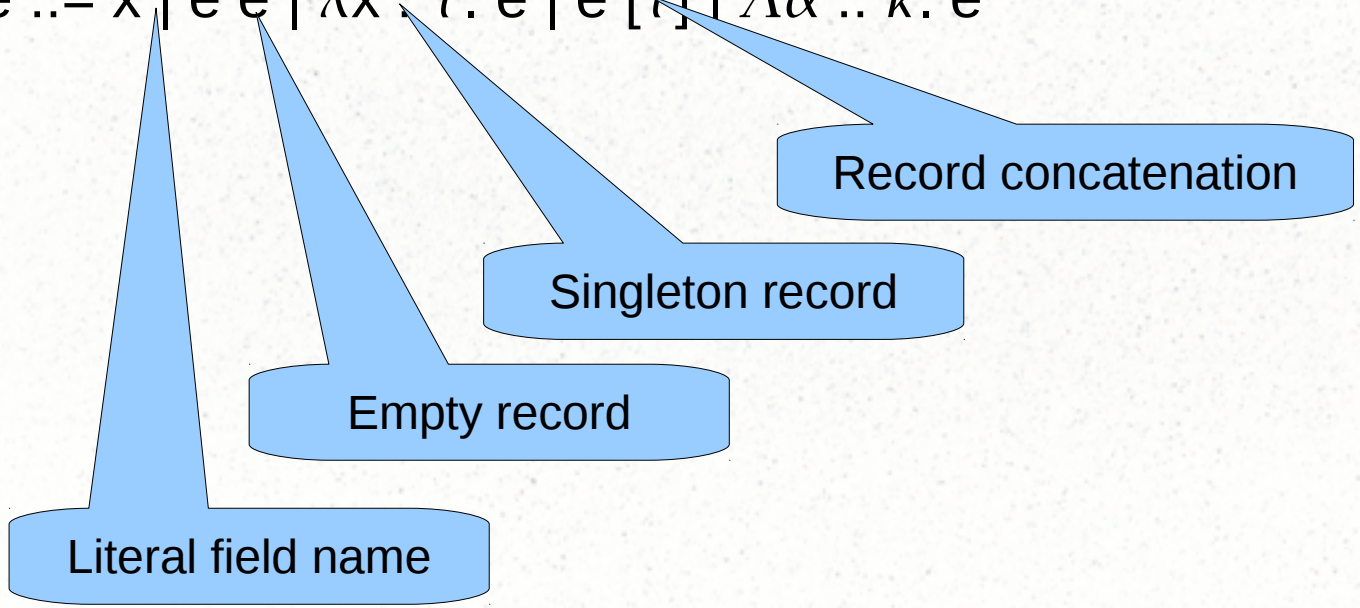
e ::= x | e | $\lambda x : \tau.$ e | e $[\tau]$ | $\Lambda \alpha :: \kappa.$ e

For type-level *record field names*

For type-level *records*

Core Language Features

κ ::= Type | $\kappa \to \kappa$
| Name | $\{\kappa\}$
τ ::= $\tau \to \tau$ | α | $\forall \alpha :: \kappa. \tau$ | $\tau\ \tau$ | $\lambda \alpha :: \kappa. \tau$
| #n | [] | $[\tau = \tau]$ | τ ++ τ | $\$\tau$ | map | $[\tau \sim \tau] \Rightarrow \tau$
e ::= x | e e | λx : τ. e | e [τ] | $\Lambda \alpha :: \kappa$. e



Core Language Features

κ ::= Type | $\kappa \to \kappa$
| Name | $\{\kappa\}$
τ ::= $\tau \to \tau$ | α | $\forall \alpha :: \kappa.\ \tau$ | $\tau\ \tau$ | $\lambda \alpha :: \kappa.\ \tau$
| #n | [] | $[\tau = \tau]$ | τ ++ τ | $\$\tau$ | map | $[\tau \sim \tau] \Rightarrow \tau$
e ::= x | e e | λx : τ. e | e [τ] | $\Lambda \alpha :: \kappa$. e

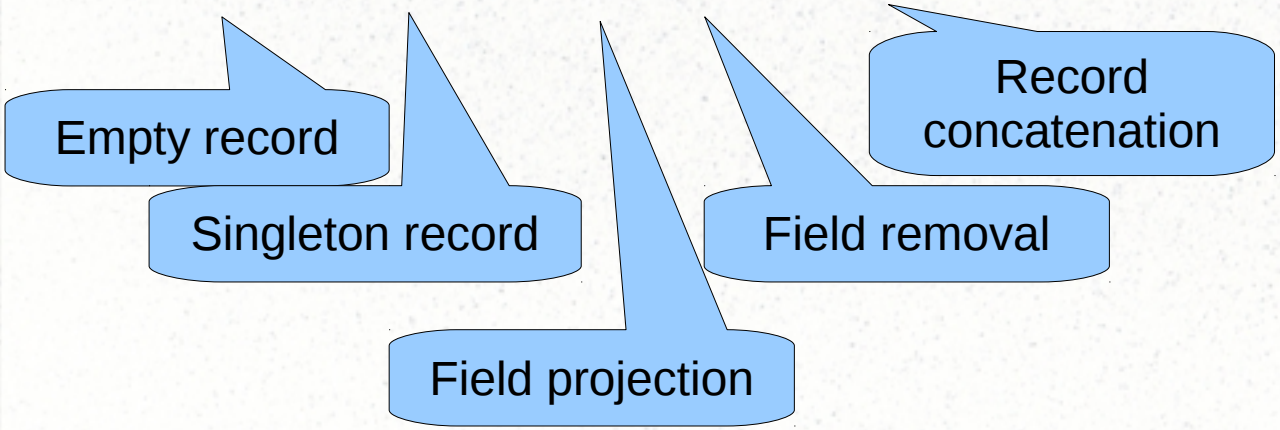
Guarded type: require that two type-level records share no field names

Apply a function to every field of a record

Convert "type-level record" to "record type"

Core Language Features

κ ::= Type | $\kappa \to \kappa$
| Name | {κ}
τ ::= $\tau \to \tau$ | α | $\forall \alpha :: \kappa . \tau$ | $\tau\ \tau$ | $\lambda \alpha :: \kappa . \tau$
| #n | [] | [$\tau = \tau$] | τ ++ τ | \$$\tau$ | map | [$\tau \sim \tau$] $\Rightarrow \tau$
e ::= x | e e | λx : τ. e | e [τ] | $\Lambda \alpha :: \kappa$. e
| {} | {τ = e} | e.τ | e – τ | e ++ e | [$\tau \sim \tau$] $\Rightarrow$ e | e !



Core Language Features

κ ::= Type | $\kappa \to \kappa$
| Name | $\{\kappa\}$
τ ::= $\tau \to \tau$ | α | $\forall \alpha :: \kappa.\ \tau$ | $\tau\ \tau$ | $\lambda \alpha :: \kappa.\ \tau$
| #n | [] | $[\tau = \tau]$ | τ ++ τ | $\$\tau$ | map | $[\tau \sim \tau] \Rightarrow \tau$
e ::= x | e e | λx : τ. e | e [τ] | $\Lambda \alpha :: \kappa.$ e
| {} | $\{\tau$ = e$\}$ | e.τ | e – τ | e ++ e | $[\tau \sim \tau] \Rightarrow$ e | e !

Guard elimination

Guarded expression abstraction: require that two type-level records share no field names

The Type-Checker is Clever

Algebraic laws are applied automatically:

- $\tau_1 \mathbin{++} \tau_2 = \tau_2 \mathbin{++} \tau_1$
- $\tau_1 \mathbin{++} (\tau_2 \mathbin{++} \tau_3) = (\tau_1 \mathbin{++} \tau_2) \mathbin{++} \tau_3$
- $\mathrm{map}\ (\lambda\alpha.\ \alpha)\ \tau = \tau$
- $\mathrm{map}\ \mathrm{f}\ (\tau_1 \mathbin{++} \tau_2) = \mathrm{map}\ \mathrm{f}\ \tau_1 \mathbin{++} \mathrm{map}\ \mathrm{f}\ \tau_2$
- $\mathrm{map}\ \mathrm{f}\ (\mathrm{map}\ \mathrm{g}\ \tau) = \mathrm{map}\ (\mathrm{f} \circ \mathrm{g})\ \tau$

Demo

Ur/Web Available At:

`http://www.impredicative.com/ur/`

Including online demos with syntax-highlighted source code