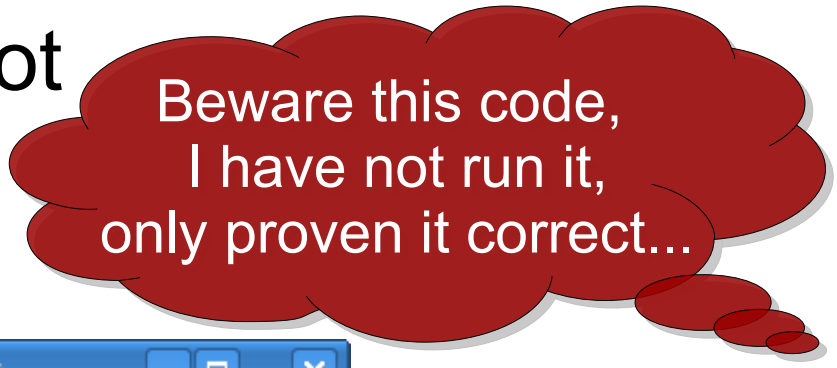
Certified Systems Development in Ynot

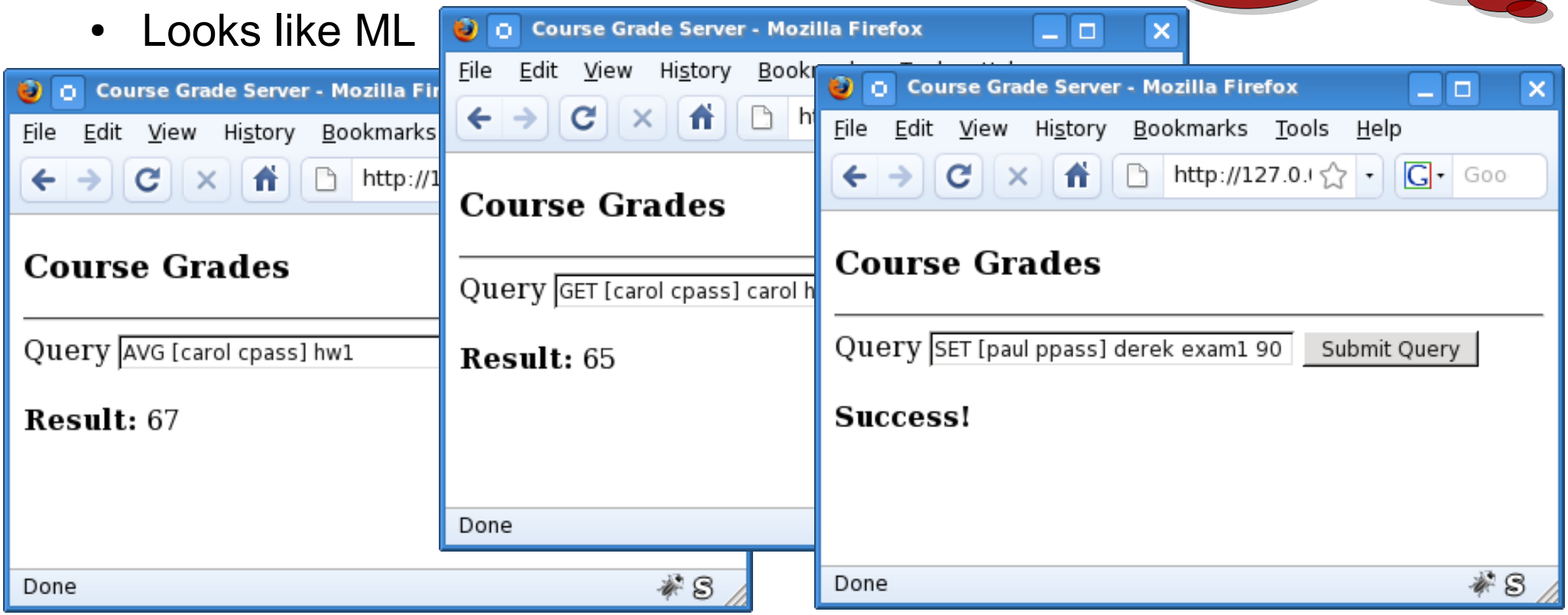
Ryan Wisnesky, Gregory Malecha, Greg Morrisett
{ryan,gmalecha,greg}@cs.harvard.edu

Special thanks to Adam Chlipala

Outline

- A 3-tier web application in Ynot
  - Provably correct
  - Runs
  - Looks like ML



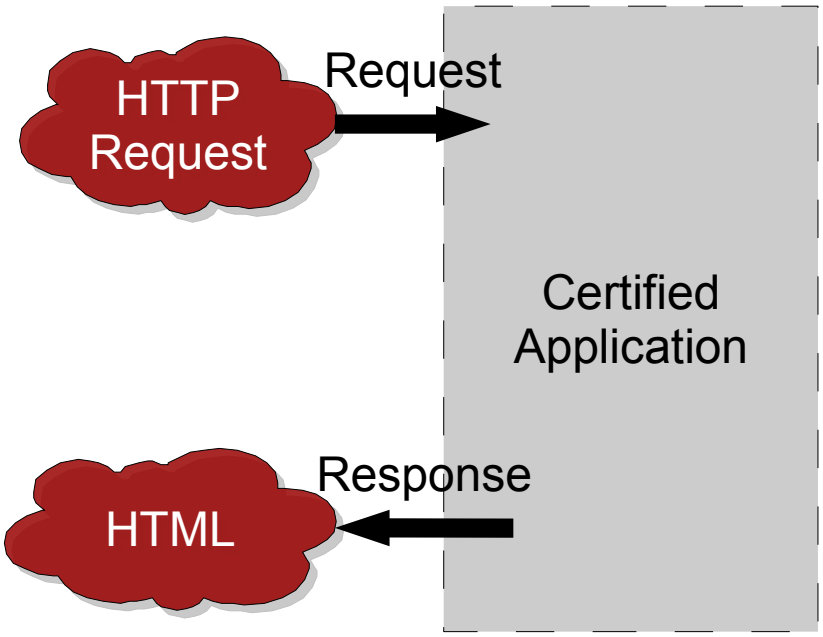


Our Web App: Gradebook

- Role-based access control

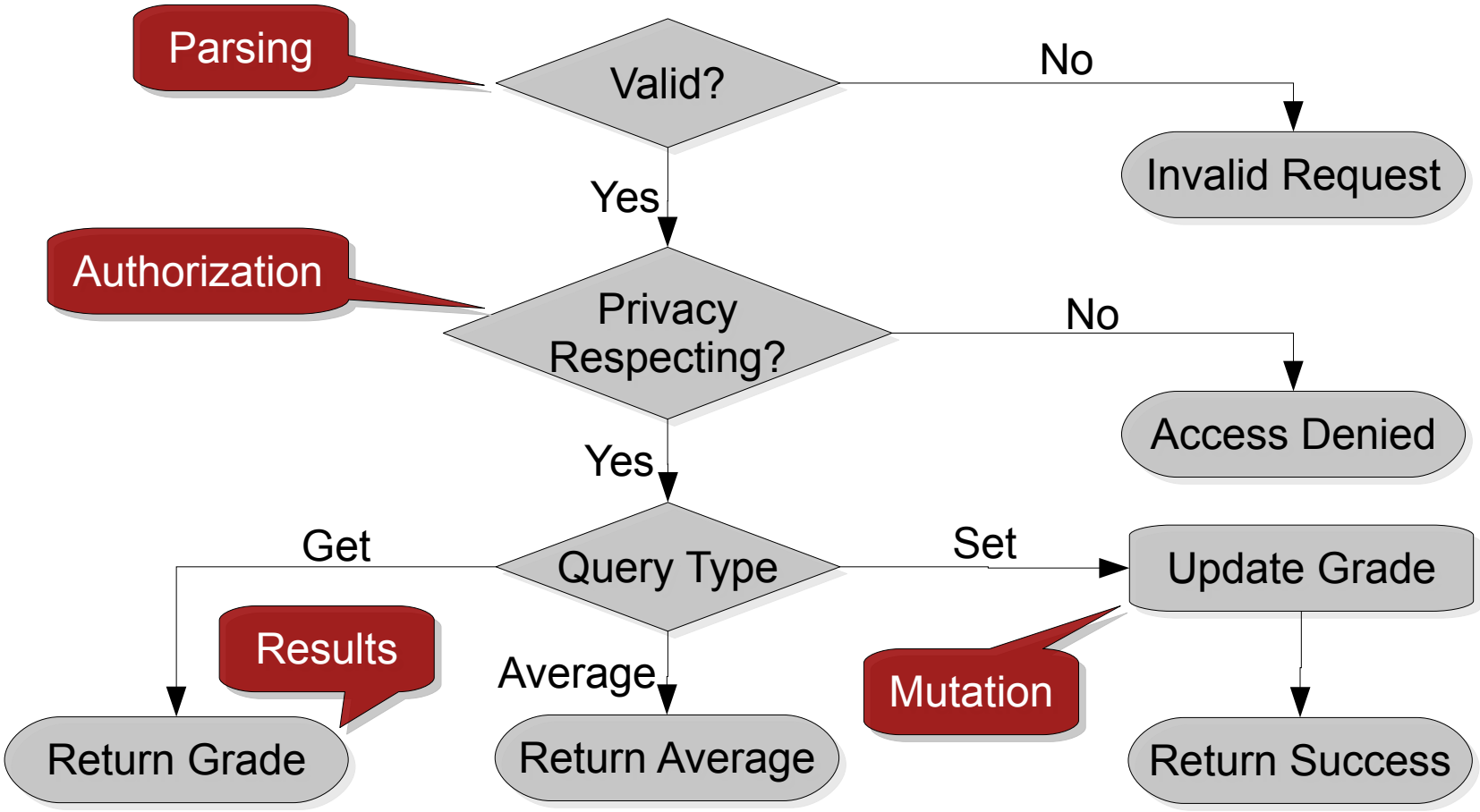
	Read	Write	Average
Students	Self	None	All
TAs	Section	Section	All
Professors	All	All	All

- Correctness depends on
  - Enforcing permissions
  - Responding correctly

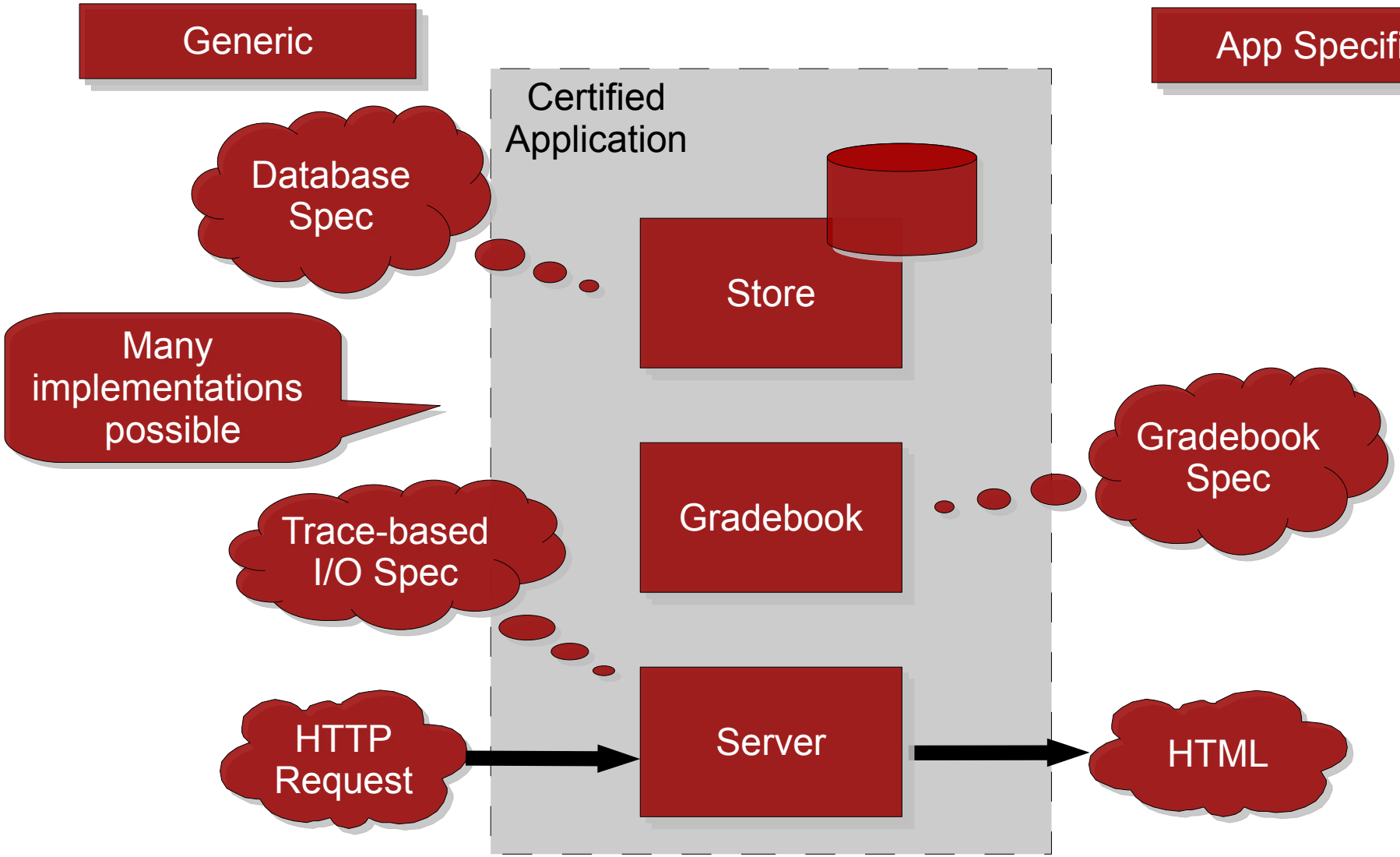


- Code, Spec, Proof all written in Coq

Specification



Architecture



OCaml


```
let swap (p1 p2 : int ref) : unit =
    let v1 = ! p1 in
    let v2 = ! p2 in
    p1 ::= v2 ;
    p2 ::= v1
```

Haskell

```
swap :: IORef Int → IORef Int → IO ()
swap p1 p2 = do v1 ← ! p1
                v2 ← ! p2
                p1 ::= v2
                p2 ::= v1
```

Monad to encapsulate effects

Explicit Sequencing

Ynot

```
Definition swap (p1 p2: ptr) (v1 v2: nat) :
  STsep (                    p1 → v1 * p2 → v2)
        (fun r:unit => p1 → v2 * p2 → v1) :=
    v1 ← ! p1 ;
    v2 ← ! p2 ;
    p1 ::= v2 ;
    p2 ::= v1.
```

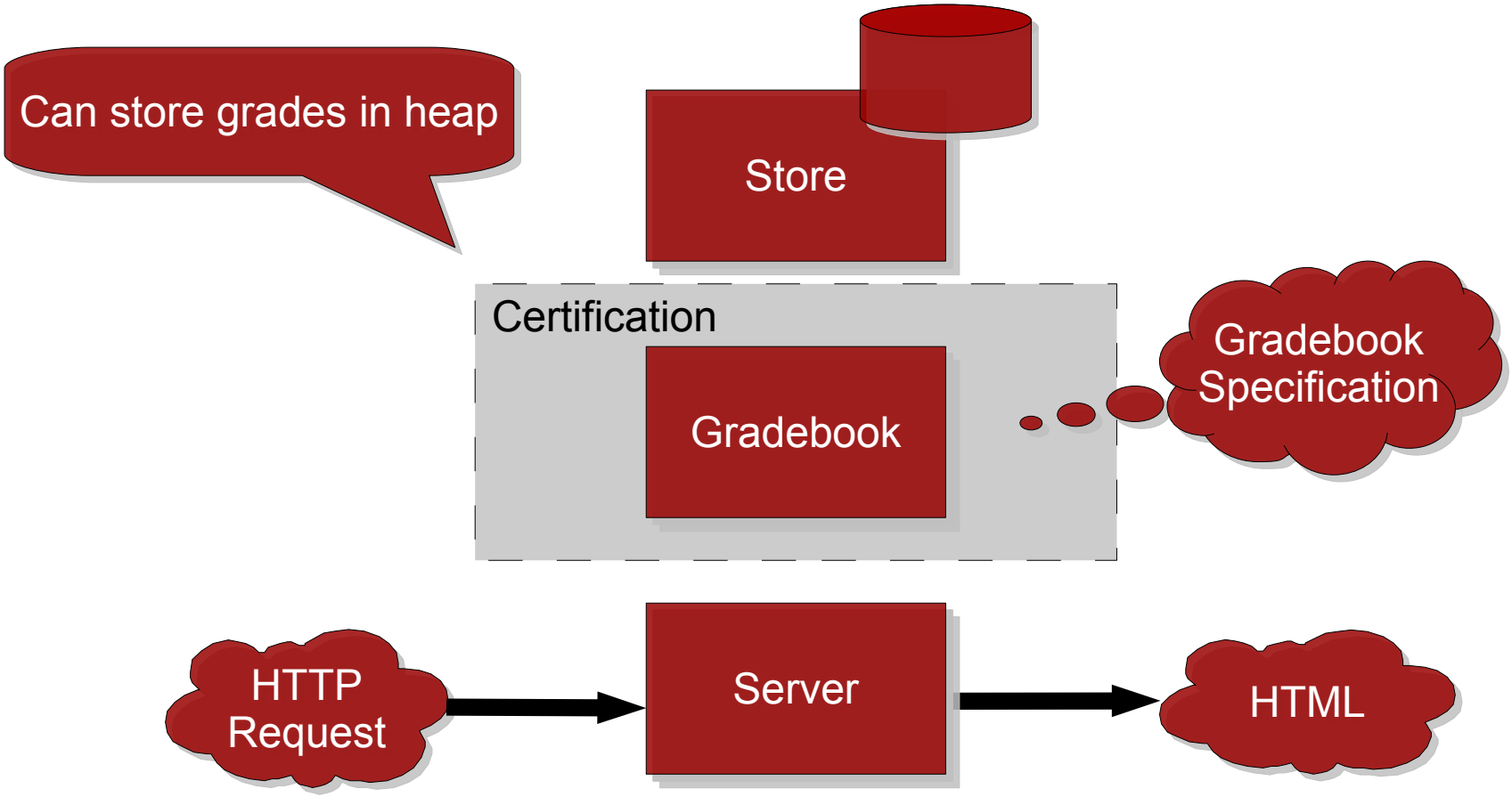
Can be made computationally irrelevant

Dependent type

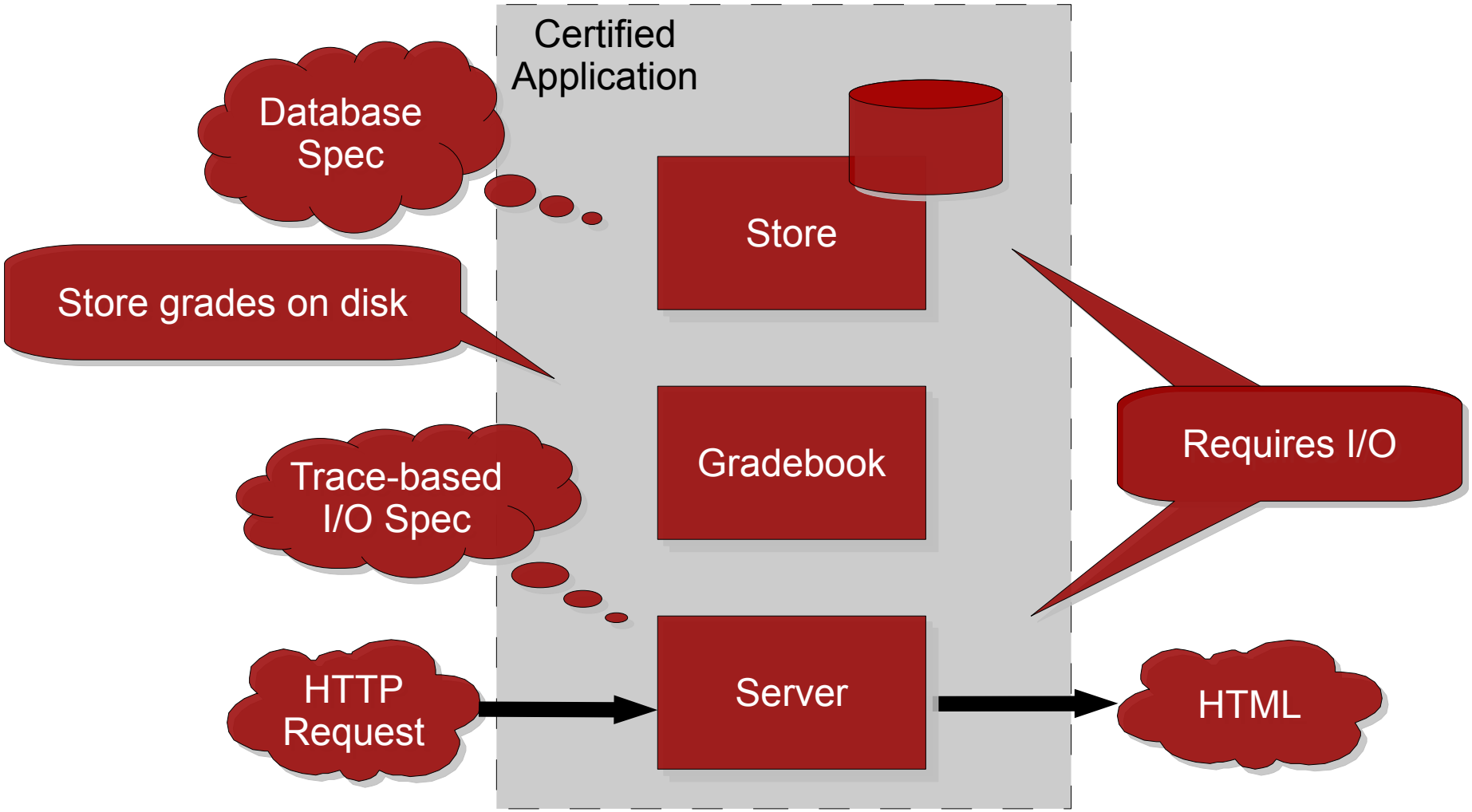
Generate proof obligations

p1 → v1 * p2 → v2
==>
p2 → v2 * p1 → v1

Architecture



Architecture: Our Contribution



Extending Ynot

Using Haskell-style types for send and recv

```
Definition echo (lsock: SockAddr) :
  STsep (                  empty)
        (fun r:unit => empty) :=
    (msg,rsock) ← recv sock ;
    send lsock msg rsock.
```

Receive and Send don't use the heap

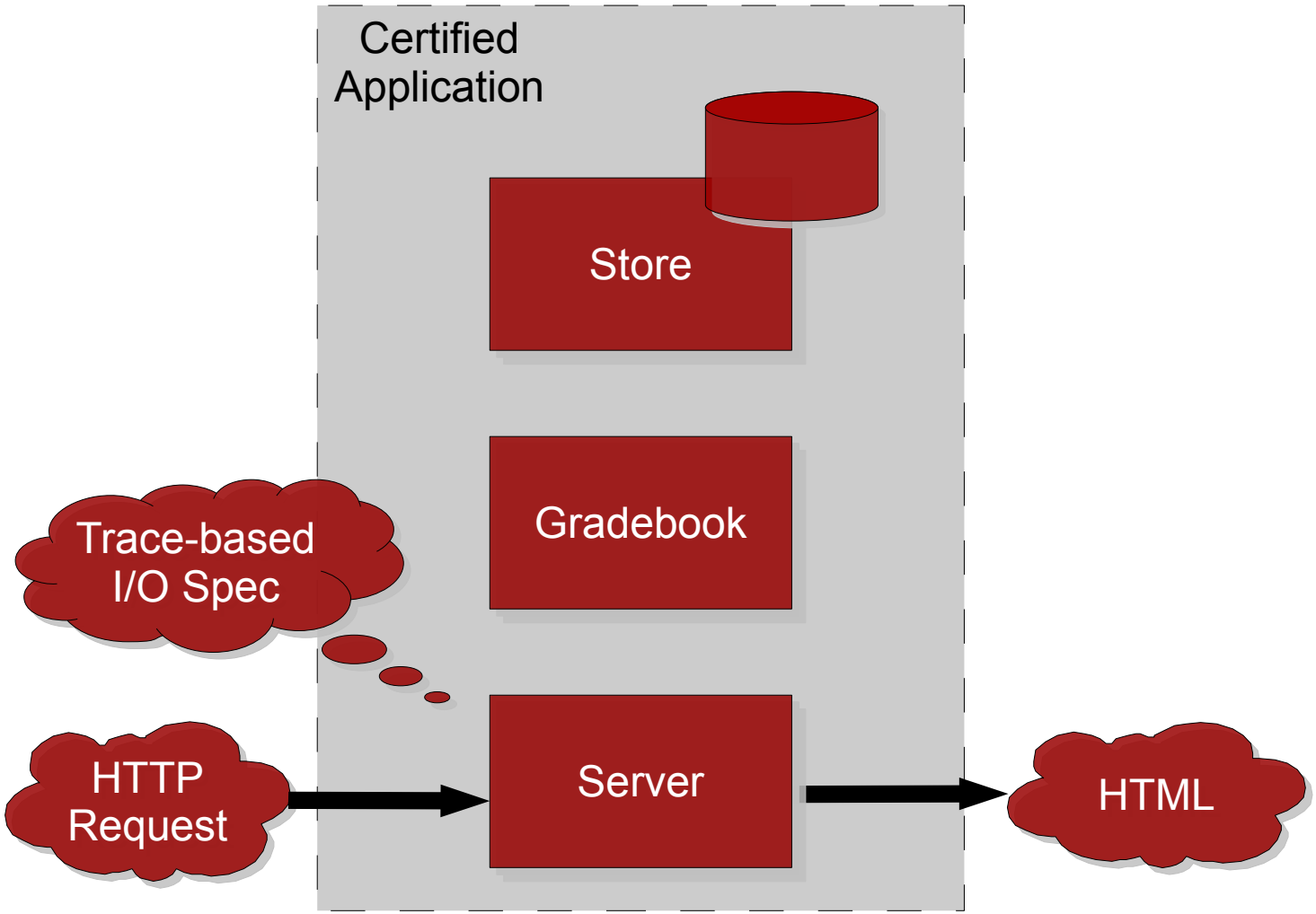
Doesn't capture the external effects

Extending Ynot

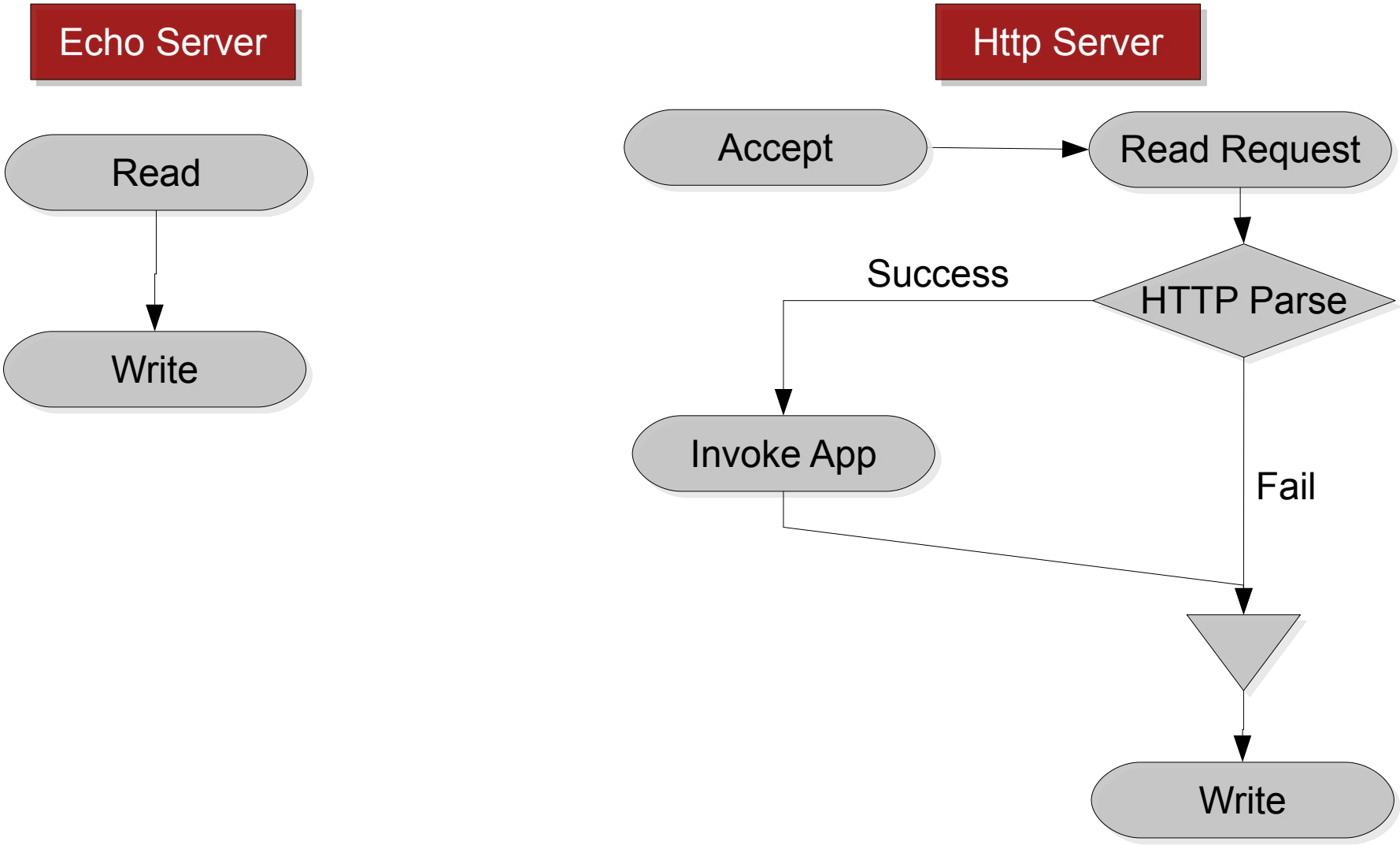

```
Definition echo (lsock: SockAddr)
   (tr : Trace) :
   STsep (traced tr)
         (fun r:unit => Exists msg rsock,
            traced (Sent lsock rsock msg ::
                    Recv lsock rsock msg ::
                    tr)) :=
     (msg,rsock) ← recv sock ;
     send lsock msg rsock.
```

Record events using traces.

From echo to Http Applications

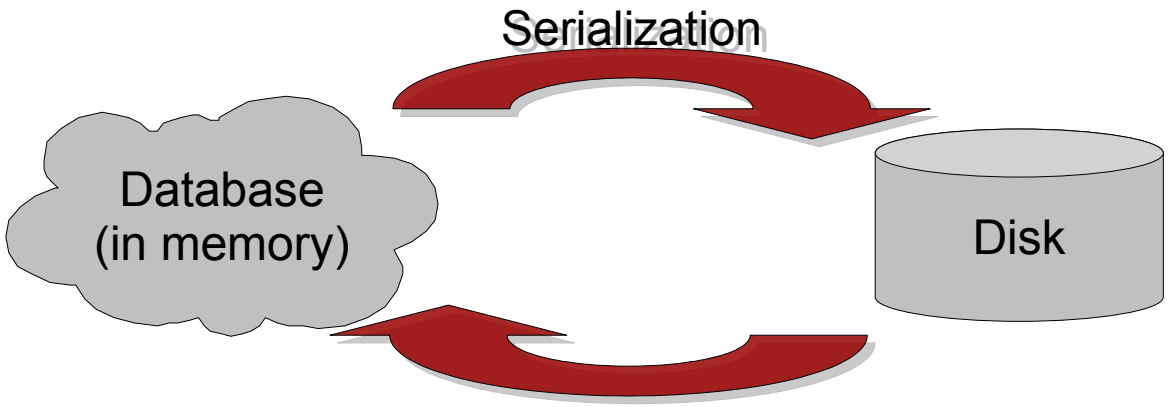


Application Server Spec

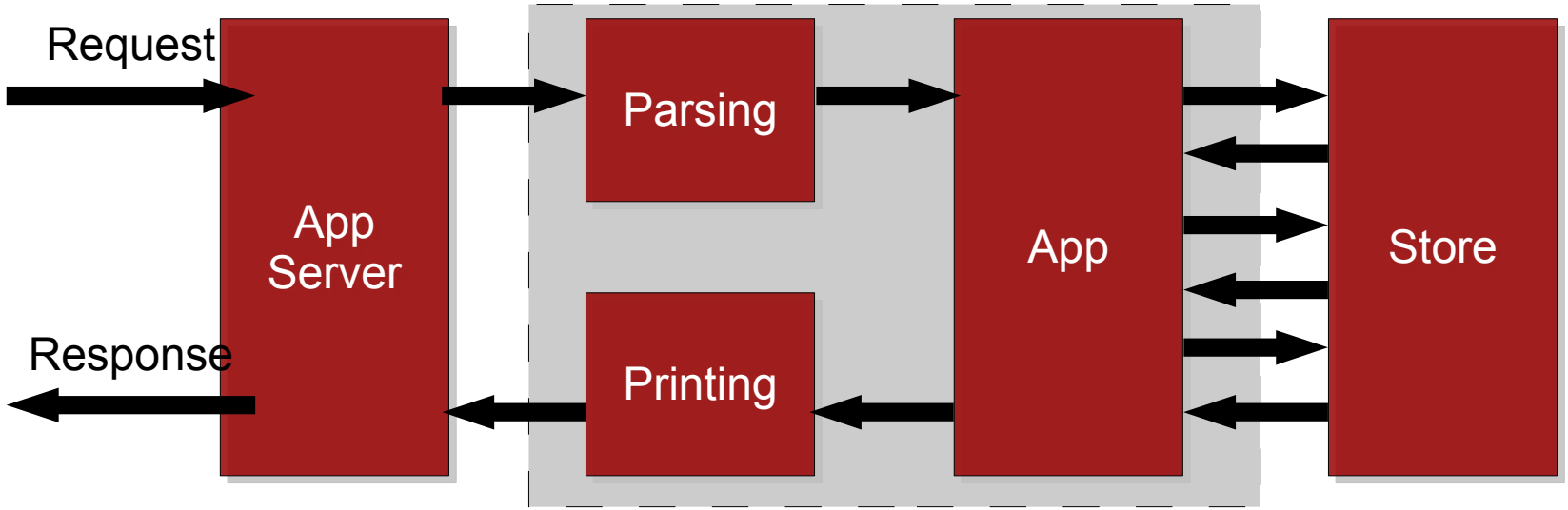


Highlights: Store

- Spec: “database-style” queries
  - Select, update, delete, insert, aggregate
- Certified mapping between grades and tuples
- Imperative implementation uses a heap-based linked-list
- Theorem: deserialize (serialize x) = x

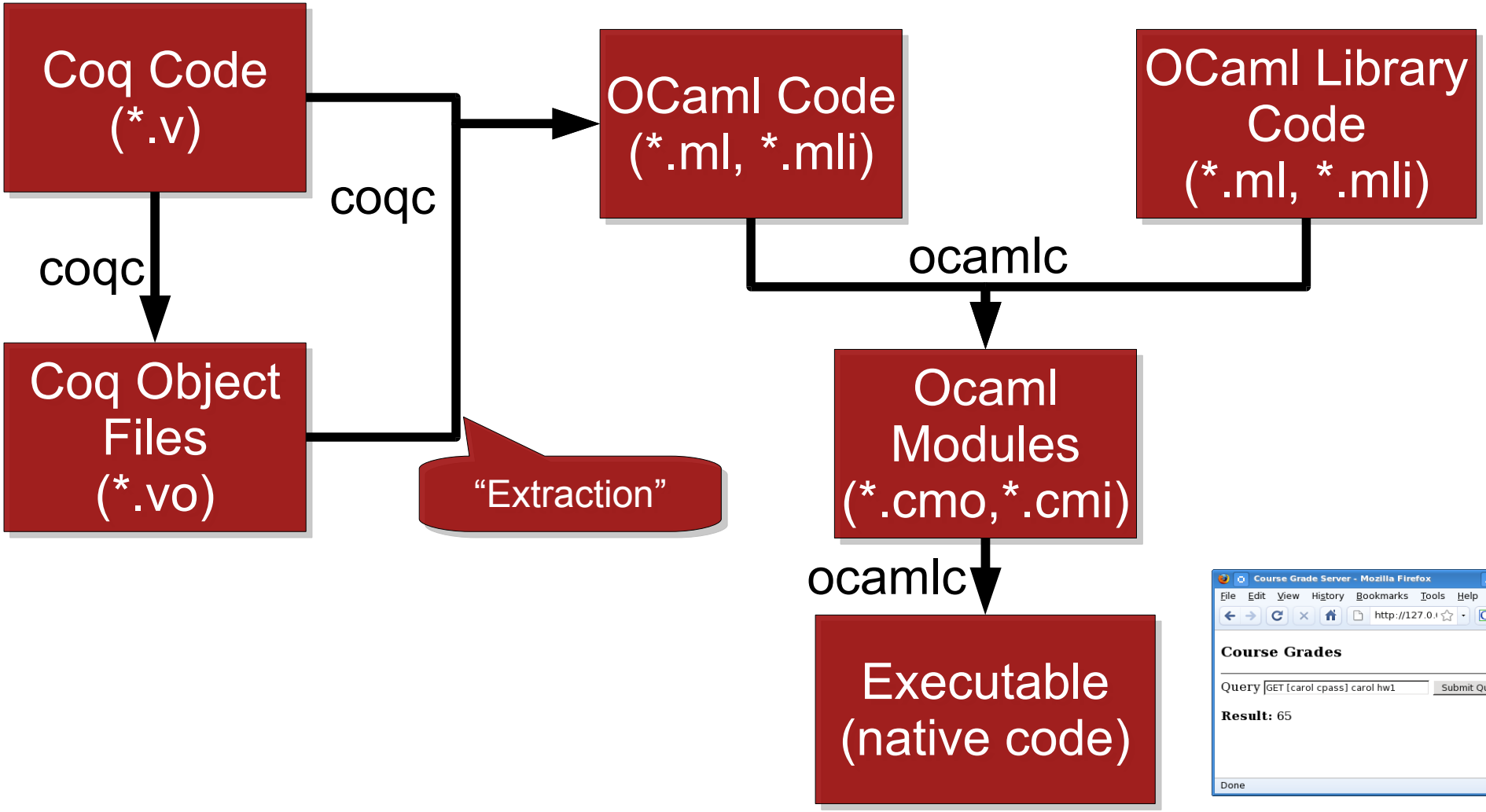


Line count



	App Server	Parsing	App	Store
Model (LOC)	414	184	231	154
Implemenation	223	269	119	113
Proofs	231	82	564	99
Overhead	1.04	.3	4.74	.88
Compile-time (m:ss)	1:21	0:55	0:32	0:23

Compilation



Conclusions

- We can build certified systems in a way like writing Haskell or ML.
- What we don't verify
  - Non-termination
  - OCaml “foreign functions”
- Demo (afterward)
  - See: http://ynot.cs.harvard.edu/

Future Directions

- Add concurrency
- More realistic database
- Resource constraints & failure modes

Persistence

- Persistance by string serialization

```
Parameter serial: Table n -> string
Parameter deserial: string -> option (Table n)
Parameter serial_deserial: forall (tbl: Table n),
  deserial (serial tbl) = Some tbl.

Parameter serialize : forall (r: db) (m: Table n),
  STsep (rep r m)
        (fun res:string =>
           rep r m * [str = serial m]).
```

Fully recoverable

Imperative implementation

Implements serial

Doesn't affect the table