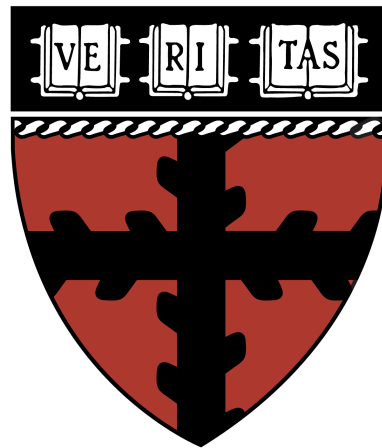


# Toward a Verified Relational Database Management System:

**Programming an RDBMS in a Proof Assistant**

Gregory Malecha, Greg Morrisett,  
Avraham Shinnar, **Ryan Wisnesky**

NEDBSummit 2010



# Goal

---

- Build an RDBMS with strong behavioral guarantees.
- ***Formal, machine-checked verification*** of:
  - Optimization
  - Compilation to physical query plan
  - Low-level (B+Tree) execution engine
- Code (and POPL 2010 paper) available at:  
<http://ynot.cs.harvard.edu>

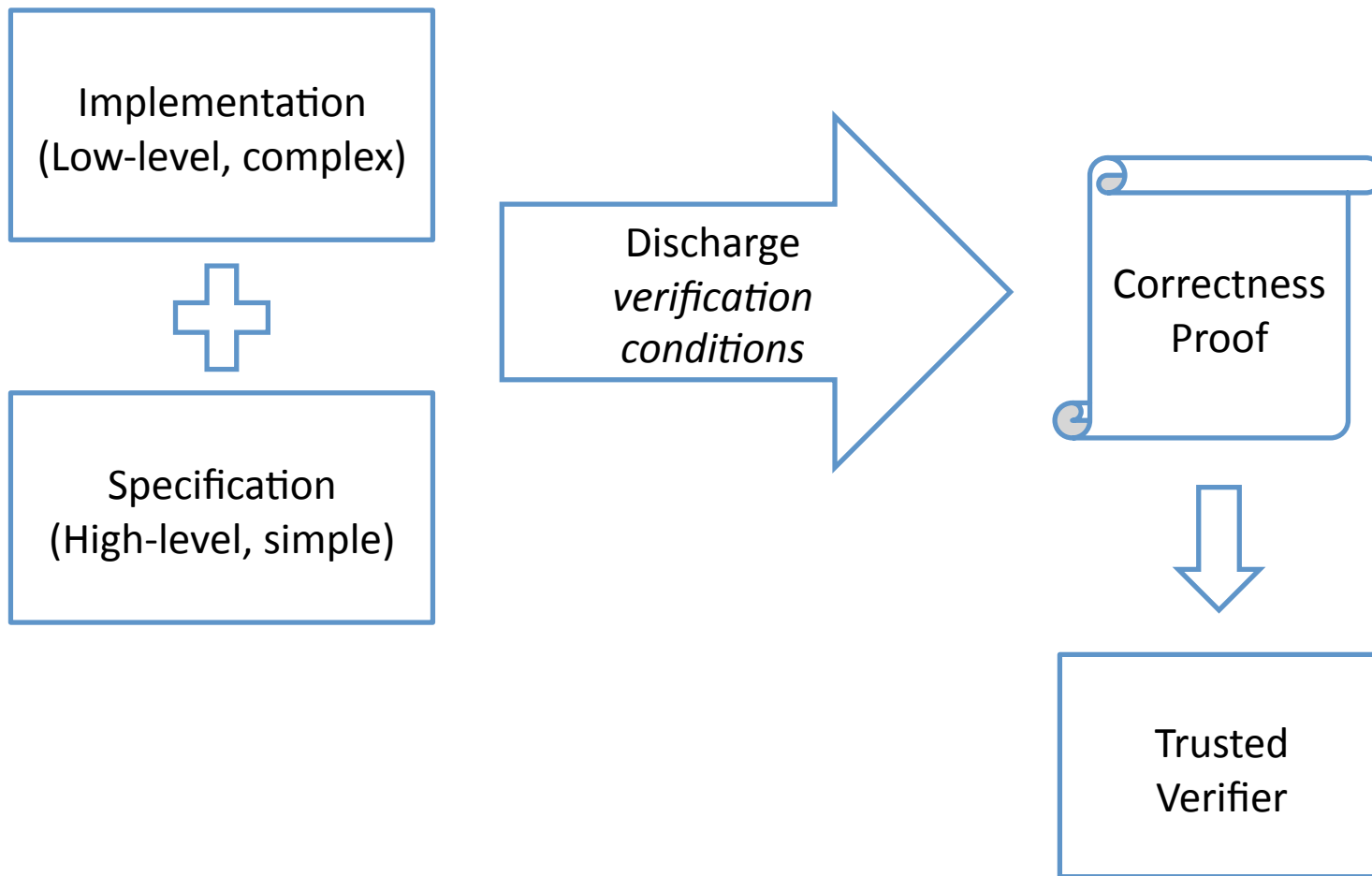
# Outline

---

- Program Verification
- Our Verified RDBMS
- Programming with a proof assistant

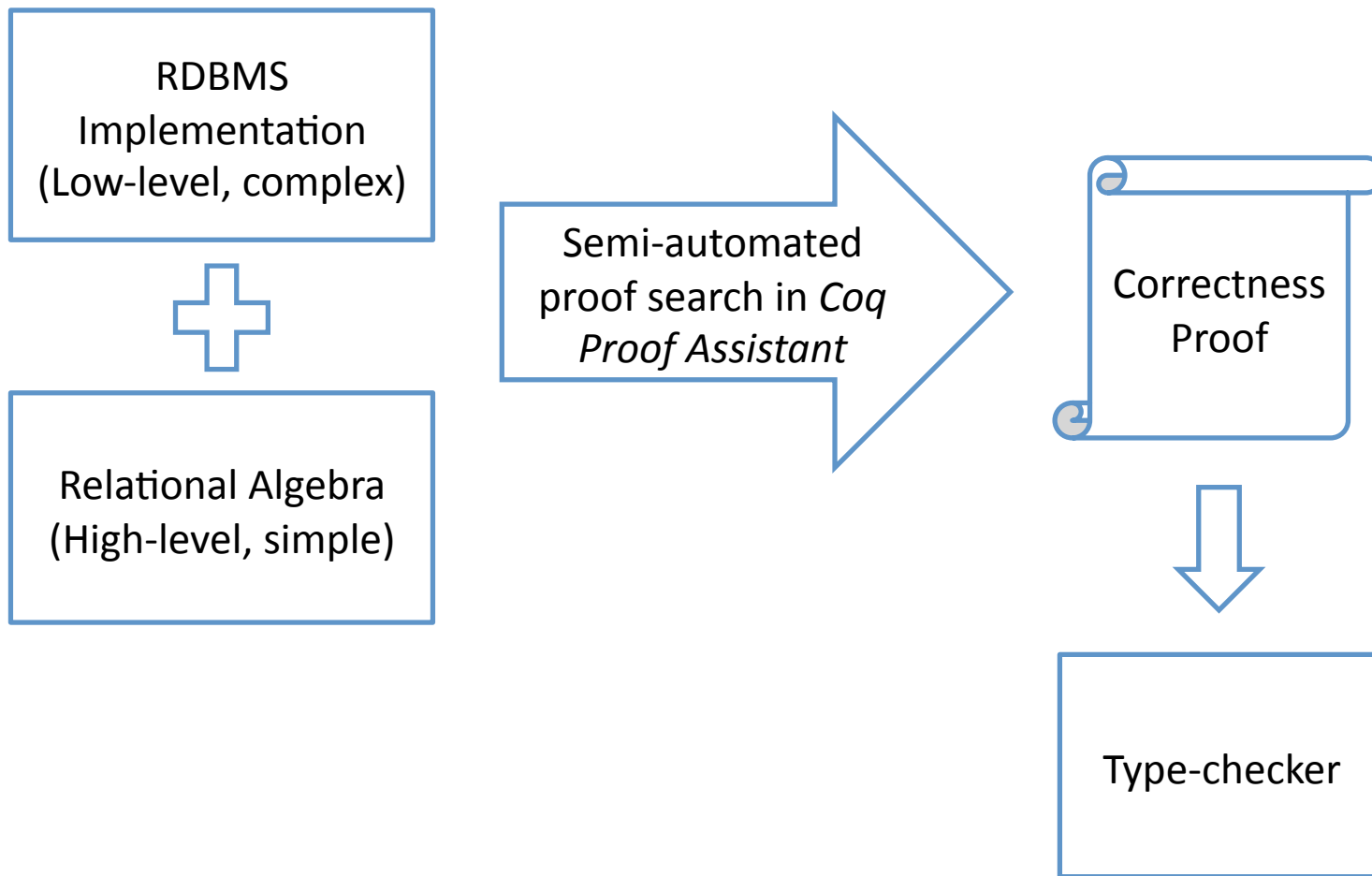
# Program Verification

---



# RDBMS Verification

---

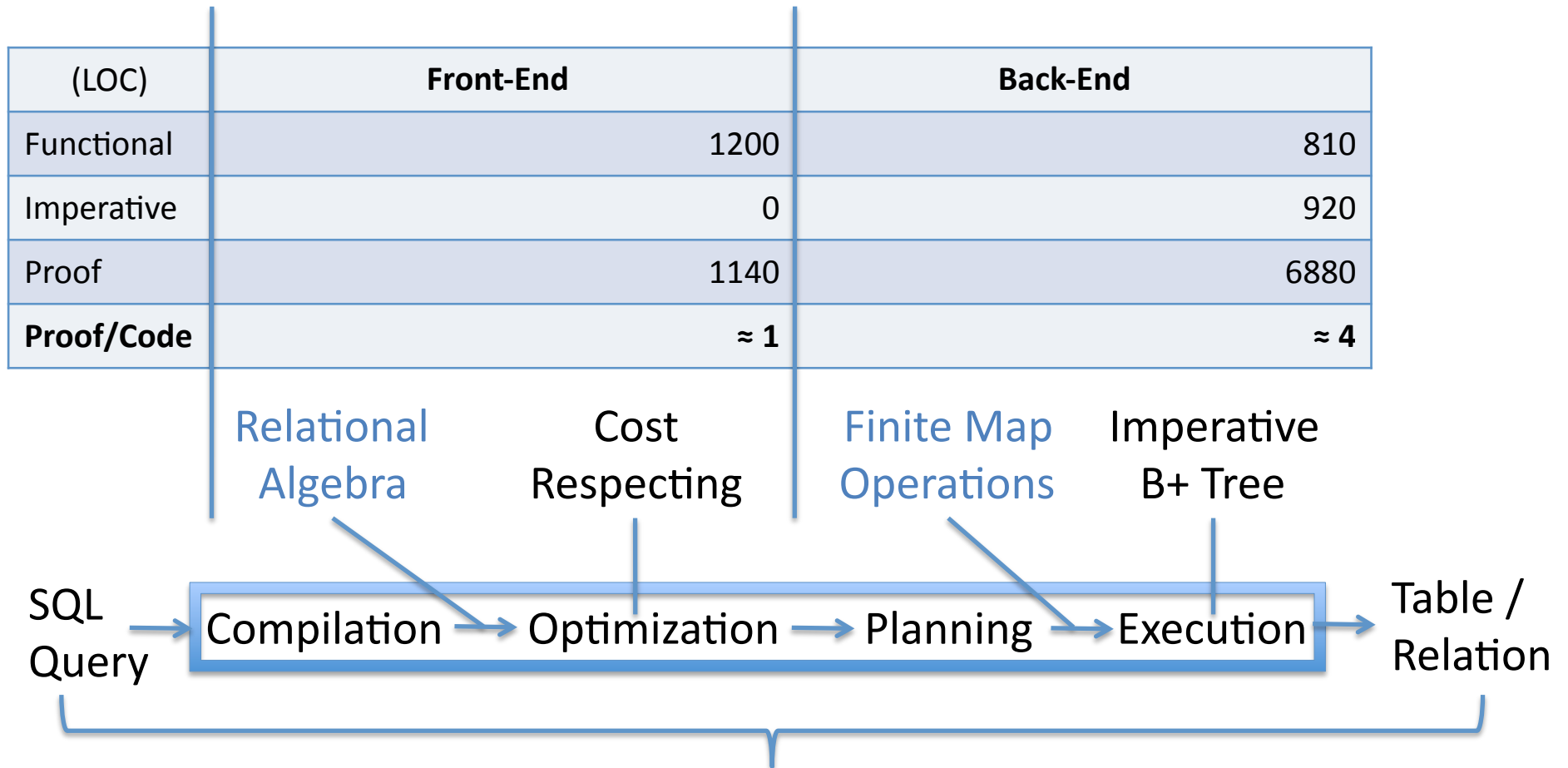


# Examples

---

- **Java VM:** bytecode obeys security policy
- **Google native client:** native code running in browser is properly isolated
- **Compcert:** compiler is semantics preserving
- **Sel4:** optimized OS implementation is correct
- **Microsoft:** device drivers do not loop forever
- **4-color theorem**

# Our RDBMS Pipeline



# The Coq Proof Assistant

---

- Core ML
- Small trusted proof checker (100s lines)
- Lightweight, pay as you go verification
- Same language for specification, implementation, and proof of correctness
- Extracts to ML/LISP/Haskell

# Append (++) in Coq

---

```
Inductive List (T: Type) : Type :=  
  | Nil   : List T  
  | Cons  : T → List T → List T.
```

```
Fixpoint ++ (T: Type) (L1 L2: List T) : List T :=  
  match L1 with  
    | Nil ⇒ L2  
    | Cons hd t1 ⇒ Cons hd (t1 ++ L2)  
  end.
```

# Theorem Proving in Coq

---

**Theorem** append\_associative:

$\forall (T: \text{Type}) (L_1 L_2 L_3: \text{List } T),$   
 $(L_1 ++ L_2) ++ L_3 = L_1 ++ (L_2 ++ L_3).$

**Proof.**

intros.

induction l1.

trivial. (\* base case \*)

simpl; rewrite IHl1; trivial. (\* inductive case \*)

**Qed.**

# Limitations of Coq

---

- Coq code must be *purely functional, terminating*.
- To use low-level imperative features like general recursion, memory, and I/O, we
  - Create *axioms* for *imperative commands*
  - Use *Hoare Logic* to reason about mutation
  - Use *Separation Logic* to reason about memory

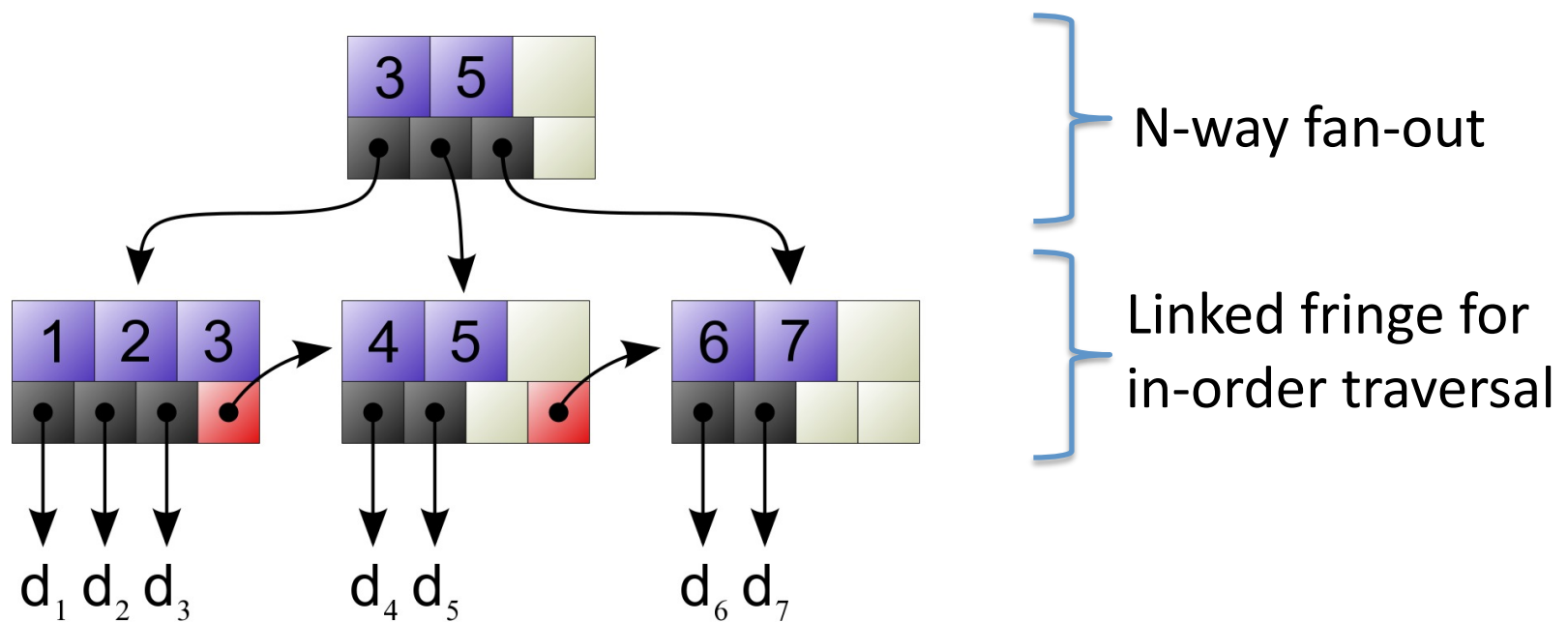
# Swap

---

```
Definition swap (p1 p2: ptr) (n1 n2: nat) :  
  Cmd (      p1 ↦ n1 * p2 ↦ n2)  
    (fun r: nat => p1 ↦ n2 * p2 ↦ n1) :=  
  v1 ← read p1  
  v2 ← read p2  
  write p1 ↦ v2  
  write p2 ↦ v1  
  return 0
```

# B+ Trees

- Generalized Binary Search Trees



# B+ Tree Invariant

---

$\text{repTree } 0 \ r \ \text{optr} \ (p', \text{ls}) \iff$   
     $[r = p'] * \exists \text{ary}. r \mapsto \text{mkNode } 0 \ \text{ary} \ \text{optr} *$   
     $\text{repLeaf } \text{ary} \ |\text{ls}| \ \text{ls}$

$\text{repTree } (h + 1) \ r \ \text{optr} \ (p', (\text{ls}, \text{nxt})) \iff$   
     $[r = p'] * \exists \text{ary}. r \mapsto \text{mkNode } (h + 1) \ \text{ary} \ (\text{ptrFor } \text{nxt}) *$   
     $\text{repBranch } \text{ary} \ (\text{firstPtr } \text{nxt}) \ |\text{ls}| \ \text{ls} *$   
     $\text{repTree } h \ (\text{ptrFor } \text{nxt}) \ \text{optr} \ \text{nxt}$

$\text{repLeaf } \text{ary} \ n \ [v_1, \dots, v_n] \iff$   
     $\text{ary}[0] \mapsto \text{Some } v_1 * \dots * \text{ary}[n - 1] \mapsto \text{Some } v_n *$   
     $\text{ary}[n] \mapsto \text{None} * \dots * \text{ary}[\text{SIZE} - 1] \mapsto \text{None}$

$\text{repBranch } \text{ary} \ n \ \text{optr} \ [(k_1, t_1), \dots, (k_n, t_n)] \iff$   
     $\text{ary}[0] \mapsto \text{Some } (k_1, \text{ptrFor } t_1) *$   
     $\text{repTree } h \ (\text{ptrFor } t_1) \ (\text{firstPtr } t_2) \ t_1 * \dots *$   
     $\text{ary}[n - 2] \mapsto \text{Some } (k_{n-1}, \text{ptrFor } t_{n-1}) *$   
     $\text{repTree } h \ (\text{ptrFor } t_{n-1}) \ (\text{firstPtr } t_n) \ t_{n-1} *$   
     $\text{ary}[n - 1] \mapsto \text{Some } (k_n, \text{ptrFor } t_n) *$   
     $\text{repTree } h \ (\text{ptrFor } t_n) \ \text{optr} \ t_n *$   
     $\text{ary}[n] \mapsto \text{None} * \dots * \text{ary}[\text{SIZE} - 1] \mapsto \text{None}$

# Review: Verified Software in Coq

---

1. Create a *simple, purely functional* application specification.
  - Example: Relational Algebra
2. Create a *sophisticated, low-level* implementation of that specification.
  - Example: Imperative B+Trees
3. Prove the implementation correct w.r.t the specification.
  - Key idea: Proofs done **interactively and semi-automatically**.

# Conclusion

---

- Verified systems software is now viable.
- Verified RDBMSs are possible.
  - 3 Ph.D. students part-time 3-6 months
  - 30 minutes to verify (3GHz Pentium D, 1GB RAM)

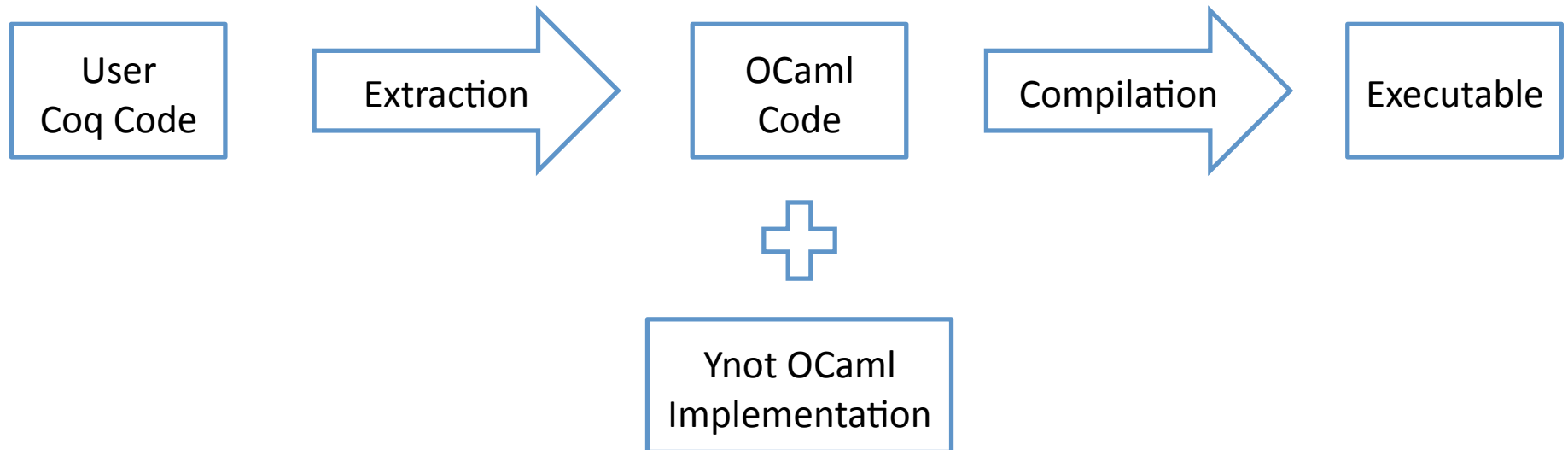
***... but still difficult, despite progress.***

- Future Work
  - Concurrency and the ACID Properties

# Questions?

# Extraction

---



# Cost Model

---

- Naïve implementation:
  - Set union:  $O(n*m)$
  - Selection:  $O(n)$
- No data statistics
- Conservative approximations
  - Selection preserves cardinality

# LOC

---

- SQLite: 65k
- Compcert: 8k Code, 24k Proof, ratio  $\approx 4$
- seL4: 47.3k Code 165k Proof, ratio  $\approx 3.5$
- Chlipala: 6.6k Code, 1.8k Proof, ratio  $\approx .3$