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

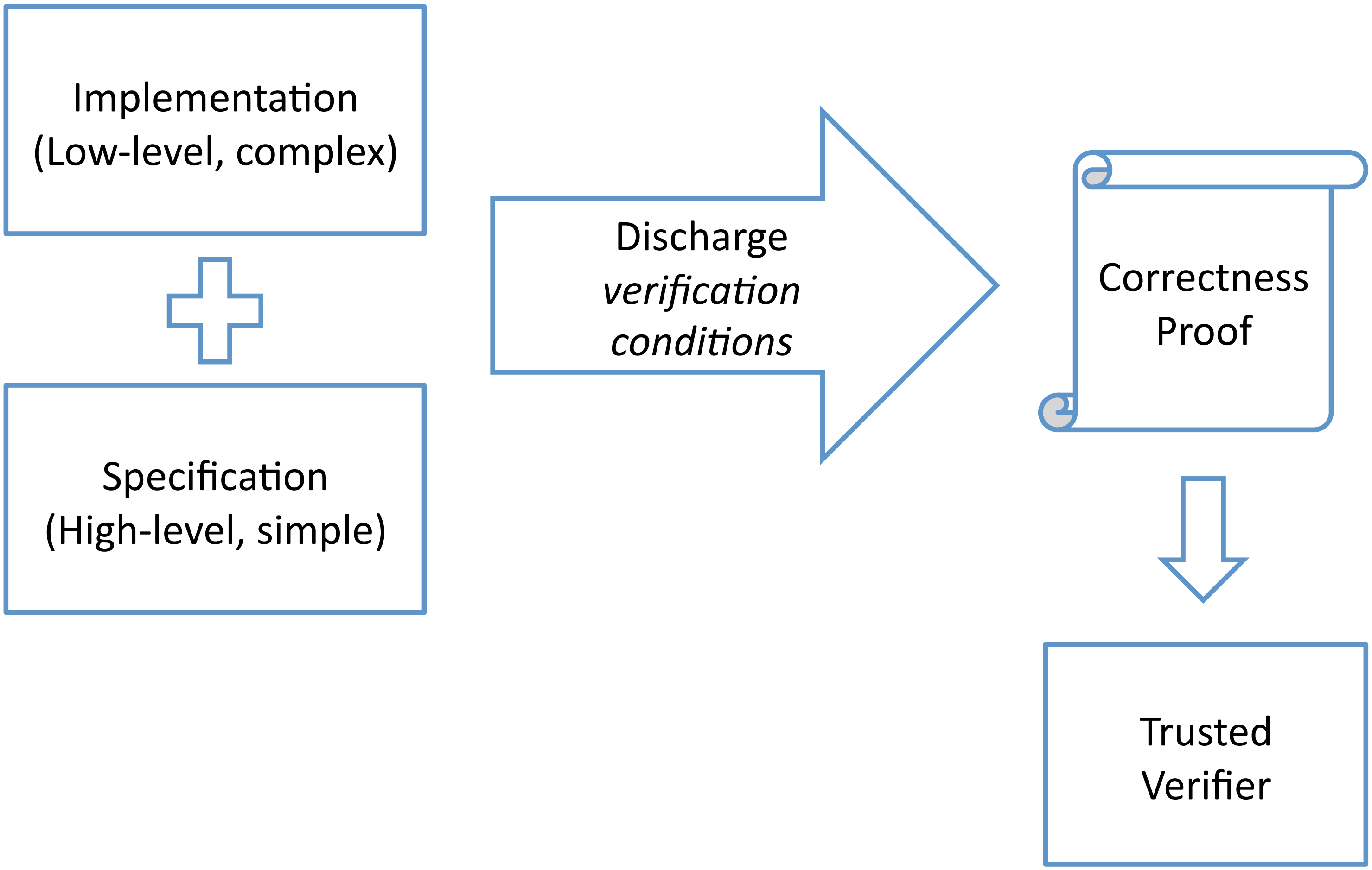
- Build an RDBMS with strong behavioral guarantees.
- ***Formal, machine-checked verification*** of:
  - Optimization
  - Compilation to physical query plan
  - Low-level (B+Tree) execution engine
- Code (and POPL 2010 paper) available at:

  `http://ynot.cs.harvard.edu`

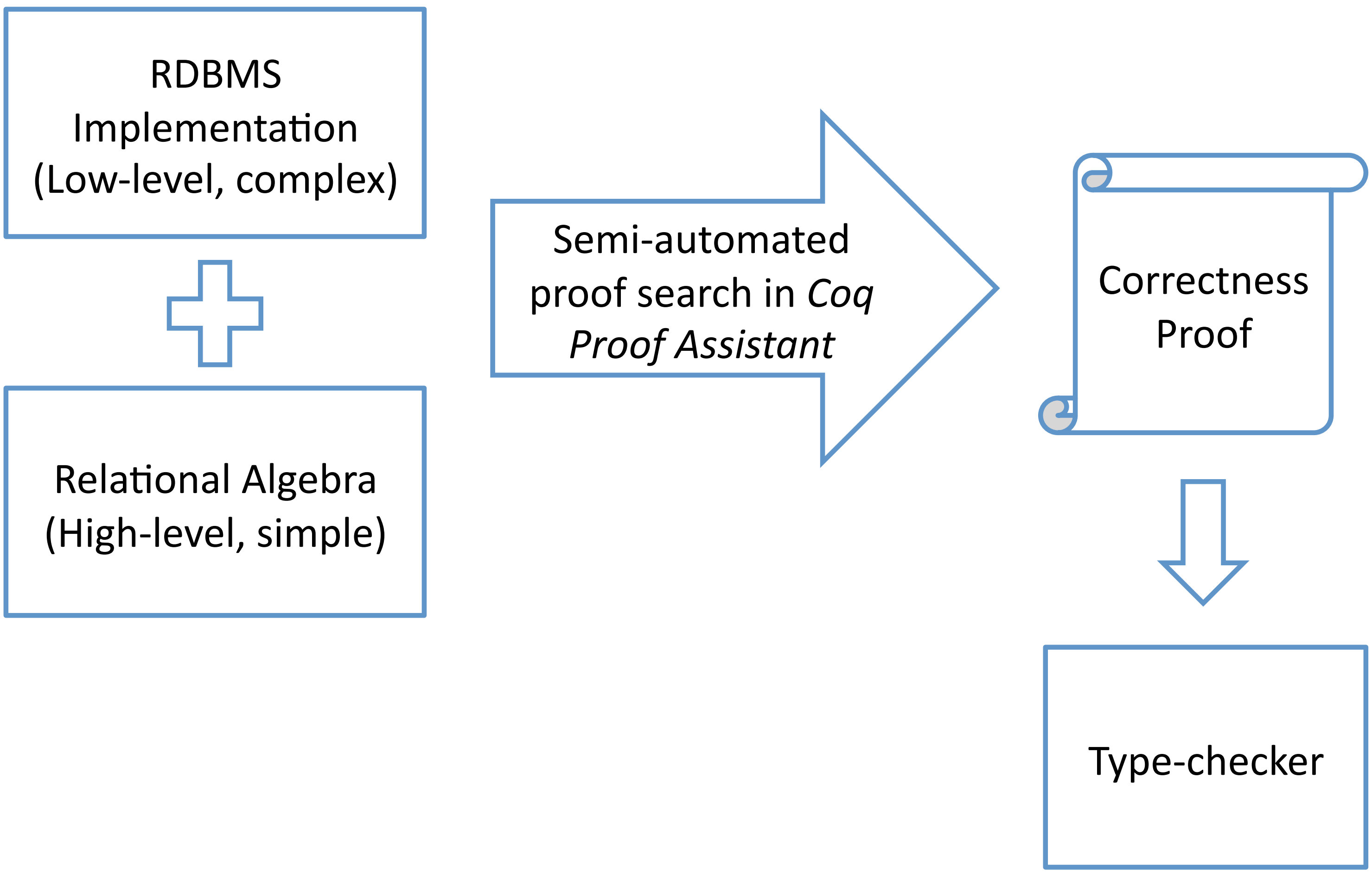
Outline

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

Program Verification



RDBMS Verification

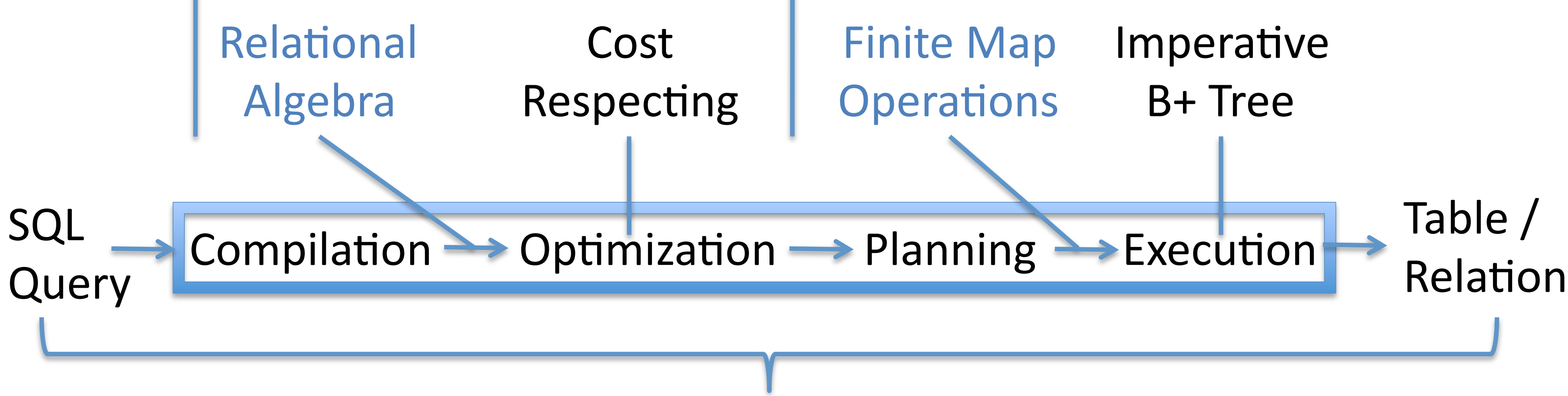


Examples

- **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

(LOC)	Front-End	Back-End
Functional	1200	810
Imperative	0	920
Proof	1140	6880
Proof/Code	**≈ 1**	**≈ 4**



The table the SQL query denotes and
the table the RDBMS returns are equal (partial correctness).

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 tl ⇒ Cons hd (tl ++ L₂)
  end.
```

Theorem Proving in Coq

```
Theorem append_associative:
 ∀(T: Type) (L1 L2 L3: List T),
     (L1 ++  L2) ++ L3 = L1 ++ (L2  ++ L3).
Proof.
 intros.
 induction l1.
  trivial. (* base case *)
  simpl; rewrite IHl1; trivial. (* inductive case *)
Qed.
```

Limitations of Coq

- Coq code must be ***purely functional, terminating.***

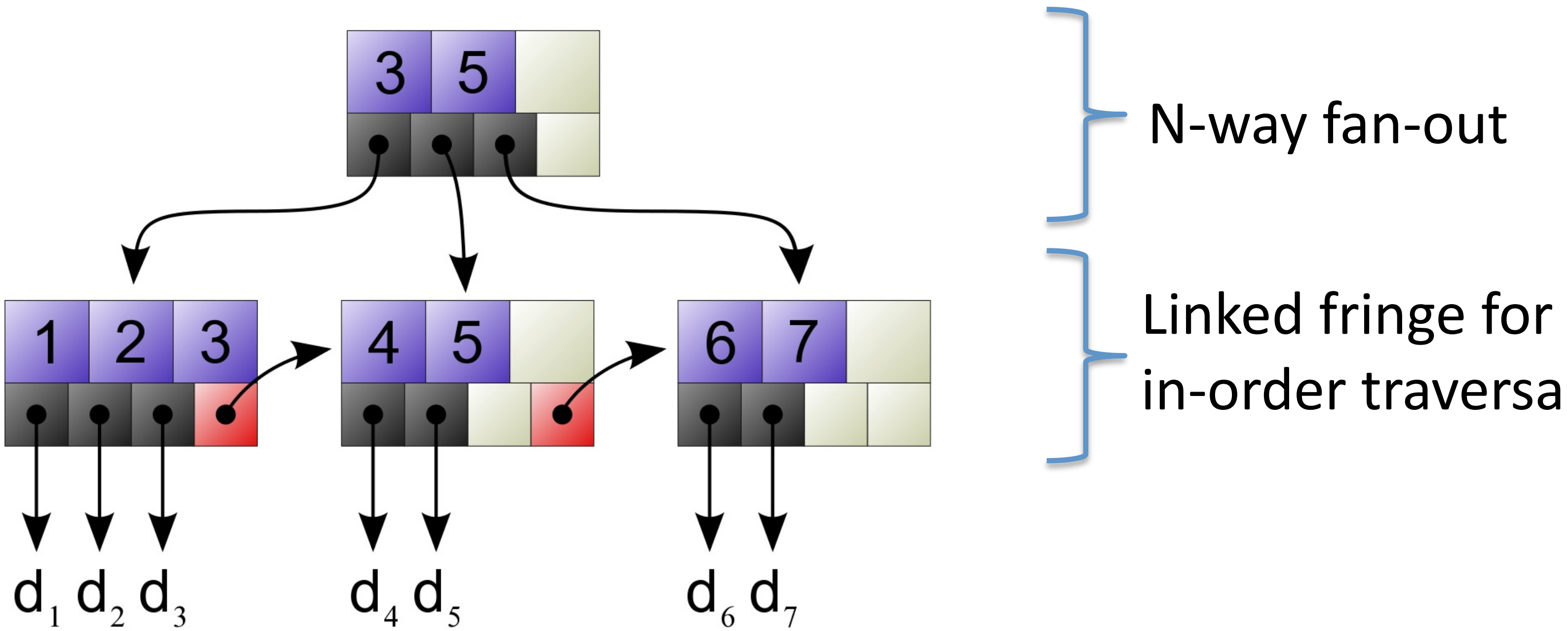
- To use low-level imperative features like general recursion, memory, and I/O, we
  - Create ***axioms*** for ***imperative commands***
  - Use ***Hoare Logic*** to reason about mutation
  - Use ***Separation Logic*** to reason about memory

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

$$\begin{aligned}
&\texttt{repTree}\ 0\ r\ optr\ (p', ls) \iff \\
&\quad [r = p'] * \exists ary.\ r \mapsto \texttt{mkNode}\ 0\ ary\ optr\ * \\
&\qquad \texttt{repLeaf}\ ary\ |ls|\ ls
\end{aligned}$$

$$\begin{aligned}
&\texttt{repTree}\ (h+1)\ r\ optr\ (p', (ls, nxt)) \iff \\
&\quad [r = p'] * \exists ary. r \mapsto \texttt{mkNode}\ (h+1)\ ary\ (\texttt{ptrFor}\ nxt) * \\
&\qquad \texttt{repBranch}\ ary\ (\texttt{firstPtr}\ nxt)\ |ls|\ ls\ * \\
&\qquad \texttt{repTree}\ h\ (\texttt{ptrFor}\ nxt)\ optr\ nxt
\end{aligned}$$

$$\begin{aligned}
&\texttt{repLeaf}\ ary\ n\ [v_1, ..., v_n] \iff \\
&\quad ary[0] \mapsto \texttt{Some}\ v_1 * ... * ary[n-1] \mapsto \texttt{Some}\ v_n * \\
&\quad ary[n] \mapsto \texttt{None} * ... * ary[SIZE-1] \mapsto \texttt{None}
\end{aligned}$$

$$\begin{aligned}
&\texttt{repBranch}\ ary\ n\ optr\ [(k_1, t_1), ..., (k_n, t_n)] \iff \\
&\quad ary[0] \mapsto \texttt{Some}\ (k_1, \texttt{ptrFor}\ t_1) * \\
&\qquad \texttt{repTree}\ h\ (\texttt{ptrFor}\ t_1)\ (\texttt{firstPtr}\ t_2)\ t_1\ \ * ... * \\
&\quad ary[n-2] \mapsto \texttt{Some}\ (k_{n-1}, \texttt{ptrFor}\ t_{n-1}) * \\
&\qquad \texttt{repTree}\ h\ (\texttt{ptrFor}\ t_{n-1})\ (\texttt{firstPtr}\ t_n)\ t_{n-1} * \\
&\quad ary[n-1] \mapsto \texttt{Some}\ (k_n, \texttt{ptrFor}\ t_n) * \\
&\qquad \texttt{repTree}\ h\ (\texttt{ptrFor}\ t_n)\ optr\ t_n * \\
&\quad ary[n] \mapsto \texttt{None} * ... * ary[SIZE-1] \mapsto \texttt{None}
\end{aligned}$$

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

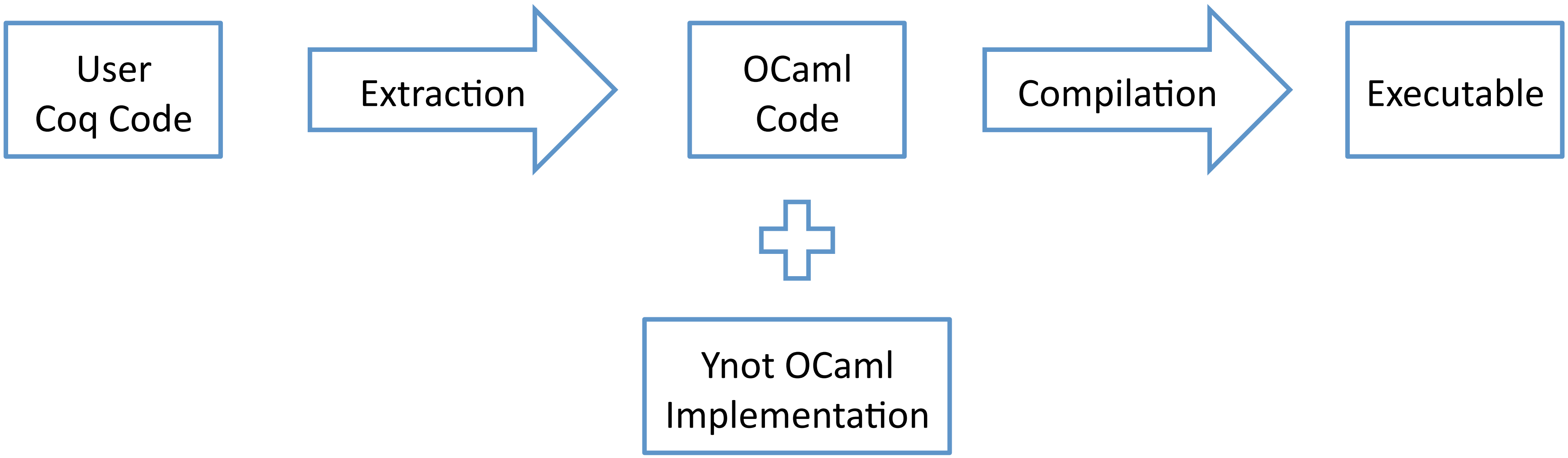
- Verified systems software is now viable.
- Verified RDBMSs are possible.
  - 3 Ph.D. students part-time 3-6 months
  - 30 minutes to verify (3GHz Pentium D, 1GB RAM)

… but still difficult, despite progress.

- Future Work
  - Concurrency and the ACID Properties

Questions?

Extraction


User
Coq Code
Extraction
OCaml
Code
Compilation
Executable
Ynot OCaml
Implementation

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 ≈ 4
- seL4: 47.3k Code 165k Proof, ratio ≈ 3.5
- Chlipala: 6.6k Code, 1.8k Proof, ratio ≈ .3