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

Implementation (Low-level, complex)

Specification (High-level, simple)

Discharge verification conditions

Correctness Proof

Trusted Verifier
RDBMS Verification

RDBMS Implementation (Low-level, complex)

Relational Algebra (High-level, simple)

Semi-automated proof search in Coq Proof Assistant

Correctness Proof

Type-checker
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

<table>
<thead>
<tr>
<th>(LOC)</th>
<th>Front-End</th>
<th>Back-End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional</td>
<td>1200</td>
<td>810</td>
</tr>
<tr>
<td>Imperative</td>
<td>0</td>
<td>920</td>
</tr>
<tr>
<td>Proof</td>
<td>1140</td>
<td>6880</td>
</tr>
<tr>
<td>Proof/Code</td>
<td>≈ 1</td>
<td>≈ 4</td>
</tr>
</tbody>
</table>

Relational Algebra | Cost Respecting | Finite Map Operations | Imperative B+ Tree

Table / Relation

SQL Query

Compilation ➔ Optimization ➔ Planning ➔ Execution

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:
\[ (T : Type) \rightarrow (L_1 \ L_2 \ L_3 : List \ T), \]
\[ (L_1 ++ L_2) ++ L_3 = L_1 ++ (L_2 ++ L_3). \]

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 \((p_1 \ p_2 : \text{ptr}) \ (n_1 \ n_2 : \text{nat}) : \text{Cmd} (\begin{array}{c} p_1 \mapsto n_1 \ast p_2 \mapsto n_2 \\ (\text{fun } r: \text{nat} \Rightarrow p_1 \mapsto n_2 \ast p_2 \mapsto n_1) := \end{array})\) :

\[
\begin{align*}
v_1 & \leftarrow \text{read } p_1 \\
v_2 & \leftarrow \text{read } p_2 \\
\text{write } p_1 & \leftarrow v_2 \\
\text{write } p_2 & \leftarrow v_1 \\
\text{return } 0
\end{align*}
\]
B+ Trees

- Generalized Binary Search Trees

![Diagram of B+ Trees]

- N-way fan-out
- Linked fringe for in-order traversal
B+ Tree Invariant

\[\text{repTree } 0 \ r \ \text{optr} \ (p', l) \iff\]
\[\begin{align*}
[r = p'] & \land \exists \text{ary. } r \mapsto \text{mkNode } 0 \ \text{ary} \ \text{optr} \ *\\
\text{repLeaf} & \ \text{ary} \ |l| \ \text{ls}
\end{align*}\]

\[\text{repTree } (h + 1) \ r \ \text{optr} \ (p', (l, s, n)) \iff\]
\[\begin{align*}
[r = p'] & \land \exists \text{ary. } r \mapsto \text{mkNode } (h + 1) \ \text{ary} \ (\text{ptrFor } n) *\\
\text{repBranch} & \ \text{ary} \ (\text{firstPtr } n) \ |l| \ \text{ls} *\\
\text{repTree} & \ (\text{ptrFor } n) \ \text{optr } n
\end{align*}\]

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

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