Mostly-Automated Verification of Low-Level Programs in Computational Separation Logic

Adam Chlipala
Harvard University
PLDI 2011
Interactive theorem-proving in higher-order logic

Classical verification with SMT solvers

The Bedrock Coq library
Decidable Theories

Interactive theorem-proving in higher-order logic

Equality + Uninterpreted Functions + Linear Arithmetic + Arrays + ...

Classical verification with SMT solvers

Complex trigger mechanism for quantifier instantiation

Complex program annotation scheme needed to produce tractable proof obligations
Solution: **Computational** specifications use standard programming features to avoid quantifiers in almost all specifications.

Solution: **Higher-Order Separation Logic** is expressive enough to allow direct use of the most natural specs.

The **Bedrock** Coq library

Complex trigger mechanism for quantifier instantiation

Classical verification with SMT solvers

Complex program annotation scheme needed to produce tractable proof obligations
A Thread Scheduler, Abstractly

Thread #1
Private memory

Thread #2
Private memory

Thread #3
Private memory

Shared Memory

A Thread Scheduler, Concretely

Thread #1
saved PC
saved SP
next

Stack #1

Thread #2
saved PC
saved SP
next

Stack #2

Thread #3
saved PC
saved SP
next

Stack #3

Shared Data Structure
What does correctness mean?

“∀ sets of threads with specifications, written in terms of local and shared heap areas, the scheduling library satisfies all of the specs.”

Example: spec for `yield()` function

Definition $\text{yieldS} := \text{st} \rightarrow \text{Ex fr, Ex ginv, Ex invs, Ex root, susp ginv (fun sp => sep ([< sp = \text{st}\#\text{Rsp }>] * ![fr])%Sep) st#Rret} \backslash \text{codesOk ginv invs} \backslash ![ \{\text{mallocHeap 0} \} * \text{st}\#\text{Rsp} ==> \text{root} * ![\text{threads invs root}] * ![\text{ginv}] * ![fr] ] \text{ st}.$
The Coq library

Program-Independent Lemmas About Data Structures
Source Code of Program to Verify
Annotations: Invariants
Annotations: Requests to Use Lemmas

Quantifier-Free Proof Obligations That Don't Mention Program Syntax or States

SMT solver

Other heuristic provers for simpler theories...
(* Abstraction predicate for finite sets represented with unsorted linked lists *)
Fixpoint llistOk (s : set) (ls : list nat) (a : nat) : sprop :=
  match ls with
  | nil => [< a = 0 /\ s = empty >]
  | x :: ls' => Ex a', [< a <> 0 /\ s x = true >]
  * a ==> x * (a+1) ==> a'
  * ![llistOk (del s x) ls']
end.

Computational Separation Logic

Step 1. Define data structure invariants as recursive functions.

Limited form of existential quantification
Computational Separation Logic

Step 2. Prove simplification lemmas.

**Theorem** llist_empty_fwd : \( \forall s \, ls \, a, \)
\( a = 0 \rightarrow \) llisetOk \( s \, ls \, a \implies [< \, ls = nil \, \land \, s = empty >]. \)
destruct \( ls \); sepLemma.
Qed.

**Theorem** llist_nonempty_fwd : \( \forall a \, s \, ls, \)
\( a \neq 0 \rightarrow \) llisetOk \( s \, ls \, a \implies \exists x, \exists ls', \exists a', \)
\([< \, ls = x :: ls' \, \land \, s \, x = true >] \ast a \implies x \ast (a+1) \implies a' \ast \lnot llisetOk (del \, s \, x) \, ls' \, a'. \)
destruct \( ls \); sepLemma.
Qed.
Step 3. Write annotated program.

Definition linkedList := bmodule {{
    bfunction "linc" [lincS] {
        [lincS]
        While (R0 != 0) {
            Use [llist_nonempty_fwd];
            Use [llist_nonempty_bwd];
            $[R0] <- $[R0] + 1;;
            R0 <- $[R0+1]
        };;
        Use [llist_empty_fwd];
        Use [llist_empty_bwd];
        Goto Rret
    }
}}.
Computational Separation Logic

Step 4. Prove module correctness theorem.

**Theorem**  \( \text{linkedListOk} : \text{moduleOk} \text{ linkedList}. \)

\[ \text{structured}; \text{ sep}. \]

Qed.

Bedrock tactics do almost all of the work.
Why Automating Separation Logic Proofs is Easy

\[
x > 5 \land x < 7 \implies x < 7 \land x = 6
\]
Why Automating Separation Logic Proofs is Easy

\[ \text{llist}(ls_1, a_1) * \ldots \]

\[ \text{Implies} \]

\[ \ldots * \text{llist}(ls_1, a_1) \]
A Simple Algorithm

1. Use annotations to expand formulas.
2. Symbolically execute block in “pre” formula.
3. Match “pre” and “post” formulas by crossing out equal parts.
Variable standing for thread #1's invariant over its private heap

\[
\text{threadInv1} \times \text{threadInv2} \times \ldots \times \text{threadInv2} \times \ldots \times \text{threadInv1}
\]

Implies
Higher-Order Implications Are Easy, Too

Assertion that every thread's invariant is satisfied

\[ \text{allOk}(\text{invs}) \ast \ldots \ast \text{allOk}(\text{invs}) \]

Implies

\[ \ldots \ast \text{allOk}(\text{invs}) \]
Frame Rule for Free

Unification variable, which may be instantiated with any formula
## Implemented Case Studies

<table>
<thead>
<tr>
<th>Library</th>
<th>LoC (total)</th>
<th>LoC (program)</th>
<th>LoC overhead of verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Malloc</td>
<td>267</td>
<td>71</td>
<td>2.8X</td>
</tr>
<tr>
<td>ArrayList</td>
<td>771</td>
<td>272</td>
<td>1.8X</td>
</tr>
<tr>
<td>ListSet</td>
<td>157</td>
<td>54</td>
<td>1.9X</td>
</tr>
<tr>
<td>BinarySearchTree</td>
<td>355</td>
<td>127</td>
<td>1.8X</td>
</tr>
<tr>
<td>AssociationList</td>
<td>200</td>
<td>85</td>
<td>1.4X</td>
</tr>
<tr>
<td>Hashtable</td>
<td>374</td>
<td>90</td>
<td>3.2X</td>
</tr>
<tr>
<td>Memoize</td>
<td>191</td>
<td>45</td>
<td>3.2X</td>
</tr>
<tr>
<td>AppendCPS</td>
<td>155</td>
<td>56</td>
<td>1.8X</td>
</tr>
<tr>
<td>Threads</td>
<td>225</td>
<td>67</td>
<td>2.4X</td>
</tr>
</tbody>
</table>

Last two reimplement examples from **Certified Assembly Programming** project [Shao et al.], where overhead is about **100X**.
The
Bedrock
Coq library

http://adam.chlipala.net/bedrock/