From Network Interface to Multithreaded Web Applications: A Case Study in Modular Program Verification

Adam Chlipala – MIT CSAIL
POPL 2015
January 17, 2015
Whole Program

Application .................................................. Application

Proof

Proof

Proof

Systems Infrastructure
(OS kernel, database engine, compiler, ...)

Proof
Whole Program

Systems Infrastructure

Application

Memory Management

Thread Management

Blocking IO

Thread Queues

Queues

Proof

Proof

Proof

Proof

Proof
Surprisingly few systems-verification projects have used their results to connect to proofs of applications that someone is actually running.

The modular style also doesn't seem to have been used previously to verify infrastructure serious enough to connect to real applications.

This talk: a case study doing all of the above inside Coq, using the Bedrock framework.
Deployed on Autonomous Vehicles

- Relational Database
- Server Thread #1
- Server Thread #N
- Malloc Library
- Thread Library
- Nonblocking Network Syscalls

- Implemented in a Web-services DSL with a verified compiler
- Implemented in a C-like language & verified semi-manually
Example Module Decomposition: Nested Threading Abstractions
Plan for Rest of Talk

• Basic stage-setting about this style of verification
• Fundamentals of the Bedrock framework
• Adapting to interface with unverified code in a principled way
• Three “design patterns” of general interest
  – Recursive definitions of higher-order, stateful predicates
  – Good formal interfaces for threading components
  – Modular verification of DSL compilers
• Code & performance
**Foundational:**
Verification leads to a proof checked by a general-purpose proof assistant (*Coq*, in this case). *Trusted code base* includes just the *proof assistant*, operational semantics of *assembly language*, and a *specification* for the whole program.

**Highly Automated:**
Tools should fill in most of the boring details of proofs.

**Higher-Order:**
Can use higher-order logic to state elegant & general specs.

The approach in this case study starts from separation-logic tools of past work (PLDI 2011, ITP 2014) and adds a few new tricks, while also applying them on a much larger scale than before.
Inventory of Corners Cut

- Proving functional correctness for systems code, but only data-structure shape invariants for application.
- Performance of verified server is OK, but it's not hard to do better. (Some parts simplified to make proofs easier.)
- Level of proof automation varies across components. Some proofs are fairly manual. (Overall proof-to-program ratio [~5:1] remains well below those reported in related projects [>= 20:1].)
What Should We Trust?

Module

code  spec  proof

Module System Soundness Proof

+ ... +

Closed Assembly Program + proof

Module

code  spec  proof

Compilation Toolchain, starting from assembly

Assembly Language Operational Semantics

Safety Property

Untrusted

Trusted
Theorem sllMOk : moduleOk sllM.
Proof.
  vcgen; abstract (sep hints; finish).
Qed.
Notations are hiding underlying DSL features for XML pattern-matching and generation. (Network communication is all via XML over HTTP.)

```
RosCommand "setParam"(!string "$caller_id",
                   !string "$key", !any $$"value")
```

Program defines remote procedure call entry points. Manipulates relational database (simple updates and queries).

```
Do
  Delete "params" Where ("key" = "$key");
  Insert "params" ($"key", "$value");;
From "paramSubscribers" Where ("key" = "$key") Do
  Callback "paramSubscribers"#"subscriber_api"
  Command "paramUpdate"(!string "/master", !string "$key", "$value");;
```

 Callbacks trigger calls to similar functions on other nodes.

```
Response Success
  Message "Parameter set."
  Body ignore
end
end
```

Notations are hiding underlying DSL features for XML pattern-matching and generation. (Network communication is all via XML over HTTP.)

```
Theorem Wf : wf ts pr buf_size outbuf_size.
  Proof.
    wf.
  Qed.
```

Automatic well-formedness proof establishes assumption of verified compiler.
Connecting with Untrusted Support Code

We assume that the following *nonblocking system calls* exist, abstracting a TCP/IP network interface:

```c
// Standard TCP socket operations
fd_t listen(int port);
fd_t accept(fd_t sock);
int read(fd_t sock, void *buf, int n_bytes);
int write(fd_t sock, void *buf, int n_bytes);
void close(fd_t sock);

// epoll-style IO event notification
res_t declare(fd_t sock, bool isWrite);
res_t wait(bool isBlocking);
```
Add System Calls to Operational Semantics

\[
\begin{align*}
[r.\text{Sp}, r.\text{Sp} + 16) & \in \text{ValidMem} \\
m[r.\text{Sp} + 8] & = \text{buf} \\
m[r.\text{Sp} + 12] & = \text{len} \\
[\text{buf}, \text{buf} + \text{len}) & \in \text{ValidMem} \\
r'.\text{Sp} & = r.\text{Sp} \\
\forall a \in [\text{buf}, \text{buf} + \text{len}) & \rightarrow m'[a] = m[a]
\end{align*}
\]

\[
(m, \text{read}, r) \rightarrow (m', r.Rp, r')
\]

State: (memory, program counter, registers)

Each op. sem. rule has a corresponding separation logic rule, proved sound w.r.t. the original. E.g.:

\[
\begin{align*}
\{\text{buf} \rightarrow \text{len}\} \\
\text{read}(\text{sock, buf, len}) \\
\{\text{buf} \rightarrow \text{len}\}
\end{align*}
\]
Recursive, Higher-Order, Stateful Predicates

ThreadQueue

Precondition of code:

Global Invariant:
Application-specific and seen by all threads

Other thread-local state
A Simpler Case

\[ P(p) \overset{\text{def}}{=} \exists q. \ p \rightarrow 42, \ q \land \{P(p)\} q \{P(p)\} \]

Watch out for recursive predicate definitions that are \textbf{inconsistent} when interpreted naively! E.g.:

\[ P(p) \overset{\text{def}}{=} \mu \rho. \exists q. \ p \rightarrow 42, \ q \land \\{\rho\} q \{\rho\} \]

Allow code modules to come packaged with \textit{named predicate definitions}, which can be looked up within specs. Now the funky reasoning only applies when we reason about the higher-order parts of a definition.

Bravely going ahead with \textit{general-recursive predicates}! Requires restrictive \textbf{side conditions} throughout proofs, to avoid inconsistency.

\[ P(p) \overset{\text{def}}{=} \exists q. \ p \rightarrow 42, \ q \land \text{ecOf("P")}(p) \}

See paper for: a cute trick to encode named predicates as named functions in the code to verify.
Specifying a Thread Stack

**ThreadQueue** module
Parameters: a set of *worlds*, a *global invariant* `ginv` in terms of it, and an *evolution relation* \( \preceq \)

\[
\forall w. \{\text{tq}(w, q) \ast ginv(w, q)\} \\
\text{yield(q)} \\
\{\exists w'. w \leq w' \land \text{tq}(w', q) \ast ginv(w', q)\}
\]

**ThreadQueues** module
Parameters: like above, except argument to `ginv` is *set of queues*, not just one queue

\[
\forall w, Q. \{\text{tqs}(w, Q) \ast ginv(w, Q) \land \text{inq} \in Q \land \text{outq} \in Q\} \\
\text{yield(inq, outq)} \\
\{\exists w', Q'. w \leq w' \land Q \subseteq Q' \land \text{tqs}(w', Q') \ast ginv(w', Q')\}
\]

Use **ThreadQueue** as a submodule by *deriving its parameters from these*!

**Scheduler** module
Parameters: like above, except argument to `ginv` is hardcoded as *set of open files*

\[
\forall F. \{\text{sched}(F) \ast ginv(F)\} \text{ yield()} \{\exists F'. F \subseteq F' \land \text{sched}(F') \ast ginv(F')\}
\]
Example verification of a client application (echo server)

L tac t := try solve [ sep unf hints; auto ];
  unf; unfold localsInvariantMain; post; evaluate hints; descend;
  try match_locals; sep unf hints; auto.

Theorem ok : moduleOk m.
Proof.
vcodegen; abstract t.
Qed.

Ltac t := try solve [ sep unf hints; auto ];
  unf; unfold localsInvariantMain; post; evaluate hints; descend;
  try match_locals; sep unf hints; auto.

Theorem ok : moduleOk m.
Proof.
vcodegen; abstract t.
Qed.

Ltac t := try solve [ sep unf hints; auto ];
  unf; unfold localsInvariantMain; post; evaluate hints; descend;
  try match_locals; sep unf hints; auto.

Theorem ok : moduleOk m.
Proof.
vcodegen; abstract t.
Modular Verification of a DSL Compiler

Idea: Give a feature-modular proof of the DSL compiler. Define different language features as standalone macros that should be usable independently or within other DSLs.
Performance Test #1: Static Web Server

Number of concurrent requests

Transactions/sec.

Apache
Bedrock (10 threads)
Bedrock (1 thread)
Performance Test #2: Robot Directory Server

Number of concurrent nodes

Seconds to complete test

Python

BWS
Summary: It is feasible today to verify a usable system including both infrastructure and application code, with a modular reasoning style, mostly automated proofs, and a final theorem checked in Coq with minimal trust dependencies.

Bedrock is on the Web at:
http://plv.csail.mit.edu/bedrock/
Backup Slides
The Bedrock Intermediate Language

W ::= (* width-32 bitvectors *)
L ::= (* program code block labels *)

Reg ::= Sp | Rp | Rv
Loc ::= Reg | W | Reg + W
Lvalue ::= Reg | [Loc]_{32} | [Loc]_{8}
Rvalue ::= Lvalue | W | L
Binop ::= + | - | *
Test ::= = | != | < | <=

Instr ::= Lvalue := Rvalue | Lvalue := Rvalue Binop Rvalue

Jump ::= goto Rvalue | if Rvalue Test Rvalue then goto L else goto L

Block ::= Instr*; Jump
Module ::= (L: Block)*
Verification Foundation: XCAP \cite{Ni&Shao:2006}

1. Whole programs

Basic block

- Precondition

- ____________
- ____________
- ____________
- jmp ____________

2. Modules

Code: [--- ----]

Assumptions: \{spec1\}label1, \{spec2\}label2, ...

\textbf{Correct program:}

Each precondition is true each time we reach it.

Proof: Assumptions imply no precondition violations within these blocks.

3. Linking

[---] + [---] = [---]