Trace-based Verification of Imperative Programs with I/O

Gregory Malecha, Greg Morrisett, Ryan Wisnesky

Harvard University, Cambridge, MA, USA

Abstract

In this paper we demonstrate how to prove the correctness of systems implemented using low-level imperative features like pointers, files, and socket I/O with respect to high level I/O protocol descriptions by using the Coq proof assistant. We present a web-based course gradebook application developed with Ynot, a Coq library for verified imperative programming. We add a dialog-based I/O system to Ynot, and we extend Ynot's underlying Hoare logic with event traces to reason about I/O and protocol behavior. Expressive abstractions allow the modular verification of both high level specifications like privacy guarantees and low level properties like data structure pointer invariants.

Key words: Program Verification; Separation Logic; Dependent Types; Traces; Imperative Programming; Ynot

1. Introduction

In an ideal world, web services would be specified with high-level concepts and protocols, and implementations would be proven correct with respect to these specifications. Indeed, there is already much work on verifying web systems at a variety of levels of abstraction. For example, there are formal cryptographic protocols [1], session type-systems for protocol conformance [18], taint-analyses for protecting from code injection attacks [19], and tools for automatically generating low-level data marshaling code [12]. Given the wealth of knowledge about how to verify particular semantic properties, a natural next step is to verify arbitrary properties, or even all properties of interest (functional correctness). In this paper we present a full-spectrum approach to this problem for programs that use general recursion, mutable state, and file and socket I/O. We demonstrate the practicality of our method by building, with minimal overhead, a web-based course gradebook with verified properties ranging from privacy guarantees to data structure pointer invariants.

Email address: {gmalecha,greg,ryan}@cs.harvard.edu (Gregory Malecha, Greg Morrisett, Ryan Wisnesky).

We use the Coq proof assistant [2] to state specifications, create implementations, and build proofs. The richness of Coq (and systems like it [37, 30]) enables modular specifications and compositional implementations at arbitrary and varying levels of abstraction. As a result, we can verify both high-level properties (protocol conformance, privacy guarantees, etc.) and low-level properties (data structure invariants, parsing correctness, etc.) in the same tool. Our trusted code base is small, since proofs can be independently and quickly checked by a small type-checker. In this work we have opted to create proofs interactively and semi-automatically during development, using Coq's proof scripting language [13], but our approach does not preclude external automation (e.g., using automated theorem provers to discharge proof obligations [44]).

In addition to Coq, we are using the Ynot [34] library for verified, general-purpose, higher-order imperative programming inside Coq. We add a dialog-based [21] I/O system to Ynot, and we extend Ynot's underlying Hoare logic with event traces [5] to reason about I/O behavior. Executable programs are generated by compiling OCaml code extracted from Coq sources and statically linking it with an OCaml implementation of the Ynot axioms [10]. During this process, non-computational content such as specifications and proofs are erased, and Ynot references are compiled to OCaml references, greatly reducing runtime overhead.

Verifying the functional correctness of realistic programs has been a grand challenge in computer science since the 1970's [25]. Our work here demonstrates, through a complete example, the engineering costs and benefits associated with the functional verification of a simple 3-tiered web application, using Coq as a programming and proving environment. Our approach is unique (and, we believe, compelling) in combining an extremely expressive, Turing-complete dependently typed programming language with aggressive proof automation for a trace- and separation-based Hoare logic. Other approaches are discussed in Section 6.

1.1. The Gradebook Application

Our web-based course gradebook allows students, teaching assistants, and professors the ability to read, edit, and statistically aggregate grades in a web-browser in a privacy-respecting way. We use a traditional three-tiered web application architecture with role-based privacy, a persistent backend data store, an application logic layer, and a presentation component [35]. A diagram of the application is shown in Figure 1.

We specify the store using a purely functional implementation of a minimal subset of SQL, including basic select, project, update, insert, and delete commands. We have implemented an imperative store using a pointer-based data structure, but this detail is isolated from the rest of the system by higher-order separation logic [40, 38].

The application logic specifies the behavior of the gradebook using high-level domain-specific concepts like grades, assignments, and sections, and defines the protocol that makes up a valid web transaction. For example, the specification states that students cannot query each other's grades. Imperative implementations are proven correct with respect to this model.

To users, the gradebook application appears as a regular HTML-based website. The application server parses HTTP requests by compiling a PEG grammar [16] for HTTP to a packrat parsing computation in a verified way [31].

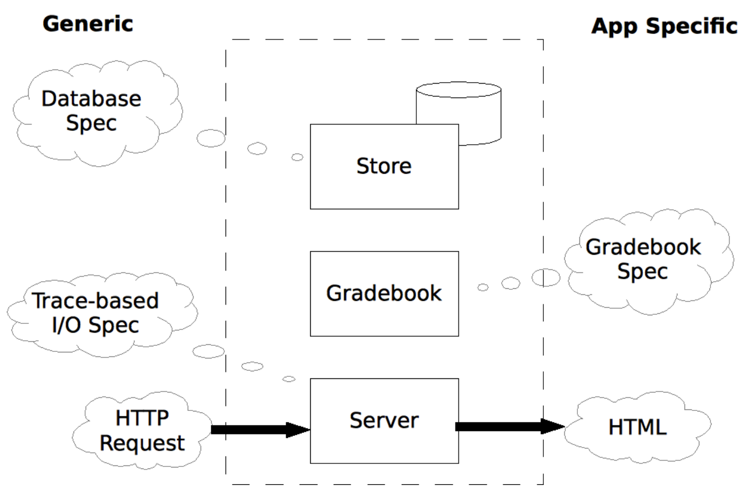


Fig. 1. The deployed gradebook application

1.2. Outline

We begin in Section 2 with a brief introduction to the salient aspects of Coq that we leverage in our development. This section is geared primarily for Coq novices and Coq experts should feel free to skip this section. In Section 3 we describe how to use Coq and Ynot to specify and implement application behaviors. We then present the verified gradebook application in Section 4, and discuss its verification overhead. In Section 5 we discuss the extraction process that we use to generate executable code. We conclude with a comparison to related tools, lessons learned, and thoughts on future work. The source code is included in the Ynot distribution at `http://ynot.cs.harvard.edu/`. For purposes of exposition we will sometimes take obvious notational liberties with Coq code.

2. Coq Background

The Coq proof assistant, usually referred to simply as Coq, is a tool for constructing programs and machine checkable proofs in the Calculus of Inductive Constructions (CIC) [2]. Coq is based on the Curry-Howard isomorphism which connects types to logical propositions and well-founded functional programs to proofs in constructive logic. This tutorial focuses on three concepts. First, we show how to define inductive data types. Next, we show how to define functions over these data types and how reason about them. Finally, we discuss axiomatic extensions to Coq.

One way to define types in Coq is using inductive definitions. For example, the Coq standard library defines natural numbers using the following inductive definition:

```
Inductive nat : Set :=
| O : nat
| S : nat → nat
```

Here, the inductive type nat is a value of type **Set**; **Set** denotes the universe of types, and thus, nat is a type. The code above defines two constructors for natural numbers. The first, on line two, states that O is a natural number which we will consider to denote 0. The S constructor on line three states that given any natural number n, S n is also a natural number, which we take to mean the successor of n, i.e. $n + 1$. Using these constructors, we could write the number 3 as S (S (S O)), though Coq provides notation that allows us to use Arabic numerals directly, e.g. 3.

We can use the same approach to define inductive predicates. For example, we can define two mutually inductive predicates that encode when a number is even or odd:

```
Inductive Even : nat → Prop :=
| EZero : Even 0
| ESucc : ∀ n, Odd n → Even (S n)
with Odd : nat → Prop :=
| OSucc : ∀ n, Even n → Odd (S n).
```

The types of Even and Odd are nat → **Prop**. This is the type of total functions from natural numbers to logical propositions. Types in the universe **Prop**, unlike **Set**, are computationally irrelevant: they can not be used to construct values in **Set**. Such irrelevance allows proofs to be ignored at runtime.

Even n (and Odd n) are the type of *proofs* that n is even (or odd). The constructors for these data types form the proof rules for a theory. EZero is the proof that states that 0 is even. ESucc (and OSucc) is the proof rule that takes two arguments, a number n and a proof that n is odd (or even), and proves that S n is even (or odd). Coq uses ∀ to denote a dependent function type where the *type* of the return value depends on the *value* of the argument. Note that n binds a variable whose type, nat, is inferred by Coq.

Coq also provides a simple **Definition** keyword as shown in the following example:

```
Definition  pair_nat  : Set := nat * nat.
Definition  list_nat  : Set := list  nat.
```

The first line defines the type pair_nat to be the type of pairs of natural numbers using the pair type constructor, $*$. Using built-in Coq notation, values of this type would be written, e.g., (1, 2). The second line uses the list type constructor to define a type of lists of natural numbers. In Coq, the empty list is denoted nil while the non-empty list is denoted by ::. For example the list 1, 2, 3 would be written 1 :: 2 :: 3 :: nil.

So far we have considered only the definition of types and propositions, but not the mechanism for computing on values or proving propositions. Both of these tasks are accomplished using a strongly normalizing (and total) functional programming language. The following three definitions are proofs that the first three natural numbers are even, odd and even respectively. Here we use the keyword **Lemma** to signify proofs, though this is simply an alias for **Definition**.

```
Lemma zero_is_even : Even 0 := EZero.
Lemma one_is_odd  :  Odd 1  := OSucc 0 zero_is_even.
Lemma two_is_even : Even 2 := ESucc 1 one_is_odd.
```

As is customary, placing two terms next to one another applies the first to the second. For example, in the second lemma, the constructor OSucc is applied to two arguments, 0 and zero_is_even.

Since there are an infinite number of natural numbers, proofs about all naturals require induction. Consider the proposition that all natural numbers are either even or odd:

```
Lemma even_or_odd : ∀ n, Even n ∨ Odd n.
```

Since we have written a definition without a corresponding body, Coq enters interactive proving mode to assist us in building it. Coq presents the following goal:

```
1 subgoal

============================
 ∀ n : nat, Even n ∨ Odd n
```

Our proof proceeds by induction on the value of n, which we can specify using the induction tactic. Coq automatically generates an induction principle for every inductive definition; in our case, this tactic applies the induction principle for natural numbers:

```
nat_ind : ∀ P : nat → Prop,
  P 0 → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

Applying the induction principle results in two new goals:

```
2 subgoals

============================
 Even 0 ∨ Odd 0

 n : nat
 IHn : Even n ∨ Odd n
============================
 Even (S n) ∨ Odd (S n)
```

We can prove the first goal by proving the left side of the disjunction and appealing to EZero. For the second goal, we consider whether n is even or odd by performing case analysis on IHn and then appealing to either ESucc or OSucc to prove either Odd (S n) or Even (S n).

At its core, this is the basic process that our verification technique; however, manually proving goals can be tedious. To ease this burden, Coq provides a scripting language, *Ltac* [13], to chain together tactics and create customized proof search heuristics that, in practice, can be used to prove many goals. Custom tactics produce Coq terms and are essential to making the verification cost of our approach manageable; we return to this point in Section 3.5.

Sometimes it is desirable or necessary to axiomatize data types and operations, rather than define them outright in Coq. Coq provides the **Axiom** keyword that we can use, for example, to axiomatize real arithmetic [1]:

```
Axiom real : Set.
Axiom real_plus : real → real → real.
Axiom real_plus_comm : ∀ a b, real_plus a b = real_plus b a.
```

[1] Coq's standard library includes a more complete, non-axiomatized formulation of real arithmetic.

Here we axiomatize `real` and `real_plus` which are respectively the type of real numbers and the addition over real numbers. We may use these two axioms to write programs with real addition, but in order to reason about our programs, we need various properties. On line three we show one such property, that `real_plus` is commutative. In the rest of the paper, we use axioms as the basis for imperative programming in Coq. Using axioms, rather than Coq definitions, allows us to extract efficient code from our imperative Coq programs; this is discussed in Section 5.

3. Trace-based Verification of Imperative Programs

We begin by discussing how we define traces in Coq in Section 3.1 and use them to specify the protocol for a simple echo server. We then discuss how to build programs that meet a trace-based specification in Section 3.2 by implementing our echo server. In Section 3.3 we extend our specifications and implementations to operations over the heap, using a counter and pointer-swapping example. We then show how to enforce modularity in Section 3.4, giving an example abstract interface for a counter. We follow up in Section 3.5 by combining the counter with the echo server to obtain an echo server that uses memory to maintain an internal counter of how many times it has echoed anything; in this section we also walk-through the human-computer interaction to discharge the generated verification conditions. We conclude in Section 3.6 by describing how more complex protocols can be composed from simpler ones.

3.1. Traces

We begin by defining traces [5] in Coq. The first primitive in our library is a socket address:

```
Axiom SockAddr : Set.
Axiom compare_sockaddr : SockAddr → SockAddr → bool.
...
```

This *axiom* tells Coq that there exists an abstract type (a `Set`, in Coq parlance) of `SockAddrs`. An alternative is to *define* socket addresses within Coq, for example as a pair of tuples (32-bit IP address, 16-bit port) of natural numbers:

```
Definition SockAddr : Set := (nat * nat * nat * nat) * (nat * nat).
...
```

We prefer to keep socket addresses abstract, and to explicitly axiomatize the behavior we require from them (that they can be decidably compared, etc). Doing so keeps our assumptions about how the underlying operating system implements sockets to a minimum. In the same way, we also introduce an abstract type of `Events` that intuitively corresponds to input and output events:

```
Axiom Event : Set.
Axiom Sent : SockAddr → SockAddr → string → Event.
Axiom Recd : SockAddr → SockAddr → string → Event.
...
```

In this paper we will mostly focus on networking Events, but we also include Events for file I/O. For simplicity, we will focus on UDP events, but we have also axiomatized TCP events.

As with SockAddrs, we could define Events rather than treating them axiomatically:

```
Inductive Event : Set :=
| Sent : SockAddr → SockAddr → string → Event
| Recd : SockAddr → SockAddr → string → Event
  ...
```

However, leaving Event abstract means that Events form an open type [27], so that users of our library can define additional I/O events without needing to modify the definition of Event. On the other hand, using an axiomatization like ours means that it is more complicated to perform explicit case analysis on Events.

We next define Traces to be a finite lists of Events:

```
Definition Trace := list Event.
```

Even though our Traces are finite, we can still express programs that run forever by using a recursion primitive (Section 3.6). We will be using inductive predicates to specify the acceptable sets of traces for our applications. For example, we define the set of acceptable traces for our echo server running using a local socket (local) as:

```
Inductive echoes (local : SockAddr) : Trace → Prop :=
| NilEchoes : echoes local nil
| ConsEchoes : ∀ remote s past, echoes local past →
  echoes local (Sent local remote s :: Recd local remote s :: past).
```

This definition expresses that the empty trace is allowable (NilEchoes), and that if some trace past is allowable, then additionally echoing back a single request is also allowable (ConsEchoes). An alternative approach would be to use more sophisticated techniques, like temporal logics [11], which we could define using parameterized Coq propositions.

3.2. Imperative Programs

We build implementations of our trace-based specifications using the Ynot library for Coq. We provide a library of networking *commands* that correspond to our primitive Events which users can use to build larger programs. The types of these commands use Traces to describe the effects that the commands have on I/O. Commands (also called imperative computations) have Cmd types; Cmd is analogous to the IO monad in Haskell and is indexed by pre- and post-conditions as in Hoare Type Theory [33]. In general, pre- and post-conditions can refer to the program's current trace and state of the heap; the post-condition is written as a function over the return value. Heaps themselves have type heap. We provide primitive commands to send a string along a socket connection, and to receive (recv) a string along a socket connection; the interface is given in Figure 2.

The tr argument in the type of send is wrapped in [ ] braces to indicate that tr is computationally irrelevant, and should be erased by the compiler when an executable is generated. Explicitly marking arguments with braces drastically decreases the runtime overhead for verification—it would not be efficient to actually pass around the program trace at runtime. One minor side effect of this way of marking computational irrelevance is

```
Definition hprop := heap → Prop.
Axiom traced : Trace → hprop.
Axiom Cmd : ∀ (pre: hprop) (T: Set) (post: T → hprop), Set.

Axiom send : ∀ (local remote : SockAddr) (s : string )
    (tr : [Trace]),
  Cmd (tr ~~ traced tr)
      (fun _ : unit ⇒ tr ~~ traced (Sent local remote s :: tr )).

Axiom recv : ∀ (local : SockAddr) (tr : [Trace]),
  Cmd (tr ~~ traced tr)
      (fun r : (SockAddr * string) ⇒ tr ~~
           traced (Recd local ( fst r) (snd r) :: tr )).
```

Fig. 2. The Networking Interface

that irrelevant variables must be explicitly unpacked inside of pre- and post-conditions, here indicated with a double tilde `~~`; outside the double tilde, `tr` has type `[Trace]`; inside `tr` has type `Trace`. This explicit unpacking ensures that the Coq type checker will prevent irrelevant parameters from being used to influence runtime behavior. Further details can be found in our previous work on Ynot [10].

The assertions that index a `Cmd` connect our programs to their specifications. For example, the type of imperative commands that conform to the previously defined echo protocol is:

```
Definition echo_iter_t local : Set := ∀ (tr : [Trace]),
  Cmd (tr ~~ traced tr * [echoes local tr])
      (fun _ : unit ⇒ tr ~~ ∃ r : Trace,
           traced (r ++ tr) * [echoes local (r ++ tr)]).
```

The `[]` notation in the pre- and post-conditions is overloaded here to indicate "pure" propositions that do not mention the heap. The `*` denotes the separating conjunction that we will explain in more detail in the next section. For now, the reader can consider it as similar to the standard classical conjunction. List concatenation is written `++`. The ∃ r : Trace construct existentially quantifies variable r of type Trace. It is easy to see that this function's type guarantees that it respects (preserves) the `echoes` predicate (invariant) on `Trace`s that we defined earlier. As such, any computation of this type when repeated forever (starting from an allowable initial state), is a correct implementation of an echo server. We can build an implementation of one echo iteration using the `send` and `recv` primitives, in a style that looks much like Haskell:

```
Definition echo ( local : SockAddr) : echo_iter_t local .
  refine (fun local tr ⇒
    x ← recv local tr <@> _ ;
    {{ send local ( fst x) (snd x) (tr ~~~
         (Recd local ( fst x) (snd x) :: tr)) <@> _ }} );
(** Proof **)
  rsep fail auto.
Qed.
```

As in Haskell, commands are sequenced through monadic binding. Intuitively, binding two computations `c1` and `c2` means running `c1` and then running `c2` using the result of `c1` as input: we write this as v ← c1; c2, and as `c1 ;; c2` when `c1`'s output is ignored. Binding requires us to prove that the pre-condition of `c2` is a logical consequence of the post-condition of `c1`. We will defer a discussion of how the correctness of this code is proved until later (Section 3.5), but for now note that we write imperative code first and then prove correctness afterward. The `{{-}}` on lines 4-5 indicates that the type of the final send may need its pre-condition strengthened and its post-condition weakened to match the overall type of `echo_iter_t`. In this example, the Coq tactic `refine` generates proof obligations that are all solved by a call to tactic rsep **fail** **auto**. The `Proof` and `Qed` statements delineate the proof script from the definition.

We have written out the intermediate state of the trace history (the fourth argument to `send`) using an irrelevant value unpacking operation `~~~`, similar to `~~`, but such states can often be inferred. Finally, the `<@>` _ notation corresponds to a use of separation logic's frame rule in which the framed heap is inferred. We will discuss this in more detail when we introduce how to reason about the heap in the next section.

3.3. Memory

In addition to performing I/O, Ynot programs can also manipulate memory. Heap memory is accessed through the traditional `new`, `read`, `write`, and `free` commands that we reason about using separation logic [40, 38]. The notation `p` ↦ `v` represents the `hprop` that the pointer (`ptr`) `p` points to `v` in the given heap. The memory commands have types:

```
Definition SepNew (T : Set) (v : T) : Cmd empty (fun p ⇒ p ↦ v).

Definition SepFree (T : Set) (p : ptr) :
  Cmd (∃ v : T, p ↦ v) (fun _ : unit ⇒ empty).

(* SepRead is also written ! *)
Definition SepRead (T : Set) (p : ptr) (P : T → hprop) :
  Cmd (∃ v : T, p ↦ v ∗ P v) (fun v ⇒ p ↦ v ∗ P v).

(* SepWrite is also written ::= *)
Definition SepWrite (T T' : Set) (p : ptr) (v : T') :
  Cmd (∃ v' : T, p ↦ v') (fun _ : unit ⇒ p ↦ v).
```

For example, when `SepNew` is run in the empty heap with argument `v`, it returns a pointer[2] to `v`. `SepFree` is the inverse: it takes a valid pointer and frees it, hence the post-condition is the empty heap. Note that `SepFree`'s type does not mean that the entire heap is empty, only that the portion of the heap referred to by the pre-condition is empty – this is characteristic of the small-footprint approach of separation logic. Pointers in Ynot are not explicitly typed, so the `SepWrite` function allows changing the type of the value pointed to by `P`. The `*` is separation logic conjunction, indicating that the heap can be split into two disjoint portions that satisfy each conjunct. `SepRead`'s type

[2] Ynot does not allow pointer arithmetic, so pointers are essentially references.

indicates that to read p, p must point to some v; the additional parameter P can be used to dependently describe the heap around p, and P is passed v as an argument.

For example, the following program swaps the values of two pointers:

```
Definition swap (p1 p2 : ptr) (n1 n2: [nat]) :
  Cmd (                 n1 ~~ n2 ~~ p1 ↦ n1 * p2 ↦ n2)
      (fun _ : unit ⇒ n1 ~~ n2 ~~ p1 ↦ n2 * p2 ↦ n1).
  refine (fun p1 p2 n1 n2 ⇒
    v1 ← ! p1 <@> (n2 ~~ p2 ↦ n2);
    v2 ← ! p2 <@> _ ;
    p1 ::= v2 ;;
    {{ p2 ::= v1 }});
(** Proof **)
  sep inst auto.
Qed.
```

The type of swap expresses that swap takes as arguments two pointers p1, p2 and two computationally irrelevant natural numbers n1, n2 such that p1 points to n1 and p2 points to n2. If swap terminates, then p1 will point to n2 and p2 will point to n1.

The swap function itself is similar to a typical pointer-swapping function but includes extra information to help us prove partial correctness. As we stated before, refine generates proof obligations, that we discharge using Ynot's built in separation logic tactic, using a call to sep inst **auto** [10]. We will once again defer describing this process, delaying it until we have seen an example with non-trivial obligations (Section 3.5). The <@> is a use of separation logic's frame rule, which allows us to describe the portion of the heap that a computation does not use. In this example, for instance, we need to know a *framing condition* that p2 points to n2 before and after p1 is read. This fact can actually be inferred automatically (and a similar condition is inferred on the next line), but we write it out here for sake of explanation.

The memory correctness properties of our implementation, such as absence of null pointer dereferences and memory leaks, are statically guaranteed at compile-time by the proofs required to invoke the Sep commands. For example, consider the following erroneous program:

```
Definition leak : Cmd empty (fun _ : unit ⇒ empty).
  refine (v ← SepNew 1 ; {{ Return tt }}).
Proof.
```

Because the heap contains 1 after the call to New but the return type of leak states that the heap should be empty, refine generates the false obligation that a heap in which v points to 1 is also a heap that is empty:

```
v ↦ 1 ⟹ empty
```

We obtain a similar behavior with traces – if we try to give a command an erroneous trace, solving the proof obligations will fail.

3.4. Modularity

We achieve modularity in Ynot by defining abstract interfaces for imperative components. This is essential for reasoning about larger programs in a compositional way.

Consider the following interface and implementation of a simple stateful counter with an increment function, `inc`:

```
Module Type Counter.
  Parameter T : Set.  (* hidden type of implementation *)
  Definition M := nat. (* public definition of the logical model *)
  Parameter rep : T → M → hprop. (* hidden heap representation *)

  (* hidden implementation *)
  Parameter inc : ∀ (t : T) (m : [M]),
    Cmd (m ~~ rep t m) (fun _ : unit ⇒ m ~~ rep t (m + 1)).
End Counter.

Module CounterImpl : Counter.
  Definition T := ptr.
  Definition rep (t : T) (m : M) := t ↦ m.
  ...
End CounterImpl.
```

Here, the `T` parameter is the hidden type of the implementation, which corresponds to a pointer in the `CounterImpl` implementation. `M` is the exposed logical model for the data structure, in this case a natural number that is the current value in the counter. The `rep` parameter relates, through an `hprop`, the state of the imperative implementation to the logical model. The `Module Type` hides everything except the logical model, providing an abstraction barrier for users of the module. This is a classic example of an *Abstract Data Type* [28].

3.5. Example: A Counting Echo Server

We can use the Counter module to implement a counting echo server. As is typical, we begin by defining a trace-based specification:

```
Inductive echoes (local : SockAddr) : nat → Trace → Prop :=
| NilEchoes : echoes local 0 nil
| ConsEchoes : ∀ remote s past n str, echoes local n past →
  str = (ntos n ++ " : ") ++ s →
  echoes local (n + 1)
     (Sent local remote str :: Recd local remote s :: past).
```

This definition is similar to the plain `echoes` definition given before except that each trace is indexed by a natural number that corresponds to the number of echoes that have occurred. This number increases on each iteration.

The type of imperative computations that implement one iteration of this specification is:

```
Definition echo_iter_t local (cnt : Counter.t) (i : [nat]) : Set :=
 ∀ (tr: [Trace]),
 Cmd (i~~ tr ~~ traced tr ∗ [echoes local i tr] ∗ Counter.rep cnt i)
     (fun _: unit ⇒ i ~~ tr ~~ ∃ r : Trace,
          Counter.rep cnt (i + 1) ∗ traced (r ++ tr) ∗
          [echoes local (i + 1) (r ++ tr)]).
```

We can easily implement a computation that corresponds to this type: [3]

```
Definition echo (local : SockAddr) (cnt : Counter.t) (i : [nat]) :
 echo_iter_t local cnt i.
 refine (fun local cnt i tr ⇒
   x ← recv local tr <@> _ ;
   n ← Counter.get cnt i <@> _ ;
   Counter.inc cnt i <@> _ ;;
   {{ send local (fst x) ((str2la (ntos n ++ " : ")) ++ snd x)
          (tr ~~~ (Recd local (fst x) (snd x) :: tr)) <@> _ }});
(** Proof **)
 rsep fail auto. (* solves 7 of 8 obligations *)
 sep fail auto; simplr.
```

Unlike our previous examples, the proof obligations generated by the counting echo sever are non-trivial. A detailed discussion of how to discharge obligations effectively can be found in [10]; here we simply want to give a flavor for how proving proceeds. The use of `refine` generates 8 subgoals, 7 of which can be discharged completely automatically by the tactic invocation rsep **fail** **auto**. After further simplification using sep **fail** **auto** and `simplr`, we are left with the following, somewhat messy, goal:

```
 local : SockAddr
 cnt : Counter.t
 x : SockAddr * string
 v : unit
 x0 : Trace
 x1 : nat
 H3 : echoes local x1 x0
 UP1 : UnpackAs (list Event) [Recd local (fst x) (snd x) :: x0]
          (Recd local (fst x) (snd x) :: x0)
 UP0 : UnpackAs nat [x1] x1
 UP : UnpackAs Trace [x0] x0
 H : x1 = x1
 H0 : echoes local x1 x0
 ============================
  empty ⟹
  [echoes local (x1 + 1)
     ((Sent local (fst x) (ntos x1 ++ " : " ++ snd x)
       :: Recd local (fst x) (snd x) :: nil) ++ x0)]
```

Here we can see that hypothesis and variable names are machine-generated, as is common in automated proving. This obligation is stated as an assertion in separation logic: in an environment where `echoes local x1 x0` holds, and the heap is empty, it must also be the case that the `echoes` predicate holds on the trace extended by `x`. This is true by definition of our `echoes` predicate, and we finish the proof by lifting the pure hprop echoes (using cut_pure) and applying standard Coq tactics to reason:

```
 simpl; cut_pure; constructor; eauto.
Qed.
```

[3] The `str2la` function converts strings to lists of characters and is defined using a simple Coq function.

The naïvely printed proof is two thousand lines long, but can be machine-checked almost instantaneously.

3.6. Application Servers

Many web systems, including our gradebook server, can be structured as computations that an application server executes repeatedly. Such web applications can be programmed using event loops in the style of dialogs [21]. At a minimum, an application iteration is defined by a progress-making, invariant-preserving Ynot command that is runnable in the initial world of an empty heap and empty trace:

```
Definition server_t (I: Trace → hprop)(pf_startable: I nil empty):=
  ∀ (tr: [Trace]),
  Cmd (tr ~~ traced tr  ∗ I tr)
      (fun r:[Trace] ⇒ r ~~ tr ~~ traced (r ++ tr) ∗
           I (r ++ tr) ∗ [r <> nil]).
```

We have implemented a number of UDP, TCP, and SSL application servers. In each case their types ensure that they only run applications that preserve some notion of partial correctness. The simplest, the `forever` server, repeats a given computation forever. The implementation of `forever` is half a dozen lines, does not require a single line of manual proof, and includes the post-condition that the server never halts (the post-condition includes `[False]`):

```
1  Definition forever : ∀ (I : Trace → hprop)
2    (B : ∀ t', Cmd (t' ~~ traced t' ∗ I t')
3                          (fun t'':[ Trace] ⇒ t' ~~ t'' ~~
4                               traced (t'' ++ t') ∗ I (t'' ++ t')))
5    (t' : [Trace]),
6    Cmd (t' ~~ traced t' ∗ I t')
7        (fun _:Empty_set ⇒ t' ~~ Exists t'' :@ Trace,
8             traced (t'' ++ t') ∗ I (t'' ++ t') ∗ [False ]).
9
10   refine (fun I B t' ⇒
11     Fix (fun t ⇒ t ~~ traced t ∗ I t)
12         (fun t (_:Empty_set) ⇒ t ~~ Exists t'' :@ Trace,
13            traced (t'' ++ t) ∗ I (t'' ++ t) ∗ [False])
14         (fun self t ⇒
15            tr' ← B t;
16            {{self (tr' ~~ t' ~~ tr' ++ t) }}
17         ) t');
18 (** Proof **)
19   sep fail auto.
20 Qed.
```

Ynot allows non-terminating recursion with an explicit Fix command shown on line 11. Since recursion invariants are generally not inferable, Fix takes the pre- and post-condition of the loop, lines 11 and 12, in addition to the function body, lines 14-16.

We have also implemented a generic HTTP server, that strips HTTP headers and passes the content on to applications. Its implementation is sketched in Figure 3, and we will use it to deploy our gradebook application, discussed in the next section.

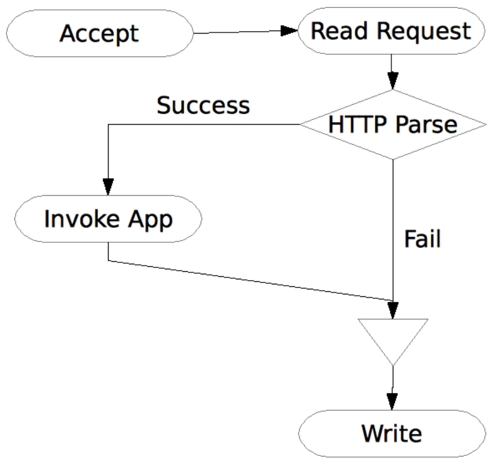


Fig. 3. Structure of the HTTP Server

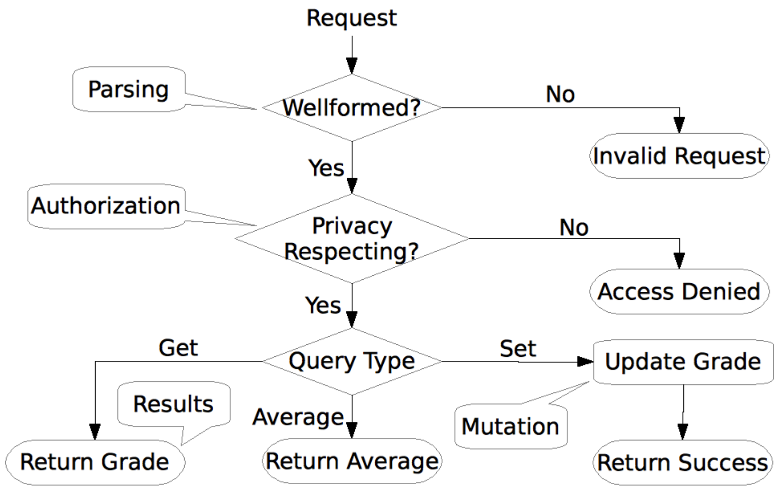


Fig. 4. Gradebook Control Flow

4. The Gradebook Application

In this section we describe the gradebook application specification, our imperative implementation, and the proof that the implementation meets the specification. We begin with the purely functional specification of the gradebook itself (Section 4.1). We then describe the entire deployed application server starting from the backend and working toward the user. We start with the data store (Section 4.2) which provides the data manipulation operations we use in our imperative implementation (Section 4.3). From there, we show how the application can be deployed using our application server (Section 4.4). We conclude with an explanation of the frontend (Section 4.5) in which we focus on parsing user requests. It is helpful to keep in mind that every imperative component is related to a purely functional model to facilitate compositionality. A diagram showing the control flow of the gradebook is shown in Figure 4.

4.1. Application Logic

In this section we define the specification of our application. We begin by defining the configuration of a course:

```
Definition ID         := nat.
Definition Section    := nat.
Definition PassHash   := nat.
Definition Grade      := nat.
Definition Assignment := nat.
Record Config : Set := mkCfg {
  students, tas, profs : list ID;
  sections : list (ID * Section );
  hashes   : list (ID * PassHash);
  maximums : list Grade
}.
```

We are using natural numbers for our basic types, but abstract types can also be used. Configurations are specified to have a number of properties; for example, all students, teaching assistants and professors must have a password and each student must belong to exactly one section. These properties are given by a Coq definition:

```
Definition correct_cfg (cfg : Config) := ∀ id,
    (In id (students cfg) ∨ In id (tas cfg) ∨ In id (profs cfg) →
      ∃ hash, lookup id (hashes cfg) = Some hash) ∧ ...
```

The actual grades are modeled by a `list (ID * list Grade)`. As with the configuration, we define a predicate `gb_inv` to ensure the integrity of the grade data. Among other things, this specifies that grade lists must always be the length of the maximums list given in the configuration, each grade must be less than or equal to the associated maximum permissible, and each student must have an entry.

The gradebook application manages a single course by processing user requests, updating the grades if necessary, and returning a response. The available commands are given by a Coq data type:

```
Inductive Request : Set :=
| SetGrade : ID → PassHash → ID → Assignment → Grade → Request
| GetGrade : ID → PassHash → ID → Assignment → Request
| Average  : ID → PassHash → Assignment → Request.
```

The meaning of a request is given by a pure Coq function `mutate` that maps a `Request`, `Config`, and `list (ID * list Grade)` to a new `list (ID * list Grade)` and one of the following responses:

```
Inductive Response : Set :=
| ERR_NOTPRIVATE : Response | ERR_BADGRADE : Response
| OK : Response | RET : Grade → Response
| ERR_NOINV : Response
```

The first four response types correspond to actual responses in the program. The application should return `ERR_NOTPRIVATE` if the request does not respect the privacy policy

```
Definition mutate (q: Command) (mm: Config * list (ID * list Grade))
  : (Status * (Config * list (ID * list Grade))) :=
  let '(cfg, m) := mm in
  if private cfg q then
    match q with
    | SetGrade id pass x a g ⇒
       if inbounds g a (totals cfg) then
         match lookup x m with
         | None ⇒ (ERR_NOINV, mm)
         | Some g' ⇒ (OK, (cfg, insert x (nth_set g a g') m))
         end
       else (ERR_BADGRADE, mm)
    | GetGrade id pass x a ⇒
       match lookup x m with
       | None ⇒ (ERR_NOINV, mm)
       | Some g' ⇒ match nth_get a g' with
                   | None ⇒ (ERR_BADGRADE, mm)
                   | Some g'' ⇒ (RET g'', mm)
                   end
       end
    | Average id pass a ⇒
       if validAssignment cfg a then
         match proj a m with
         | None ⇒ (ERR_NOINV, mm)
         | Some x ⇒ (RET (avg x), mm)
         end
       else (ERR_BADGRADE, mm)
    end
  else (ERR_NOTPRIVATE, mm).
```

Fig. 5. The functional specification of the gradebook application.

and `ERR_BADGRADE` if the user requests the grade for an assignment that does not exist. We use `OK` to denote a successful update and `RET` to return a value to the user in response to a `GetGrade` or `Average` request.

The fifth response type, `ERR_NOINV`, simplifies our definition of the `mutate` function. Technically, `mutate` is partial, i.e. it is only meaningful when the gradebook invariant `gb_inv`s holds on the gradebook data. `ERR_NOINV` is the default return value that we use when `mutate` is applied to data that does not represent a gradebook. In our implementation we prove that our gradebook implementation always respects `gb_inv`, but "over-defining" `mutate` in this way makes it easier to use with our other components. The full specification is given in Figure 5.

The function private is meant to decide whether or not a given request respects the privacy policy. By guarding the body of `mutate` with the `private` test, it is easy to see that our specification enforces privacy. However we can also easily prove properties about our policy to reassure ourselves that we encoded it correctly.

Privacy is enforced using role based access control described by the table in Figure 6. The policy is defined by the Coq predicate private also given in Figure 6.

	Read	Write	Average
Students	Self	None	All
TAs	Section	Section	All
Professors	All	All	All

```
Definition isProf (cfg: Config) (id: ID) (pass: PassHash) :=
  In id (profs cfg) ∧ lookup id (hashes cfg) = Some pass.
(* ... *)

Definition private (cfg : Config) (cmd : Request) : Prop :=
  match cmd with
  | SetGrade id pass x _ _ ⇒
    isProf cfg id pass ∨ taFor cfg id pass x
  | GetGrade id pass x _   ⇒
    isProf cfg id pass ∨ taFor cfg id pass x
    ∨ (id = x ∧ isStudent cfg id pass)
  | Average id pass _      ⇒
    isProf cfg id pass ∨ isStudent cfg id pass ∨ isTa cfg id pass
  end.
```

Fig. 6. Gradebook privacy policy.

We have proven a number of theorems about our specification, for example that `mutate` preserves `gb_inv` and do not return `ERR_NOINV` when `gb_inv` holds. To help make the proofs more tractable, we implemented a number of automated proof search tactics tailored to this model.

4.2. Data Store

The backend data store of the gradebook server is a simplified relational database. We first give a functional specification of the store, and then prove that our imperative implementation meets this specification. Logically, a `Store` is modeled by a list of `Tuple n` defined by the following Coq data type:

```
Fixpoint Tuple (n: nat) : Set :=
  match n with
  | 0 ⇒ unit
  | S n' ⇒ (nat * Tuple n')
  end.
Definition Table n : Set := list (Tuple n).
```

For simplicity our data values are only natural numbers, and we specify only a small subset of the functionality of SQL, including select, update, project, and delete. For example, selection is modeled logically by:

```
Definition WHERE := Tuple n → bool. (* “where” clause *)

Fixpoint select (wh : WHERE) (rows : Table) : Table :=
  match rows with
  | nil ⇒ nil
  | a :: r ⇒ if wh a then a :: select wh r else select wh r
  end.
```

Our purely functional model has expected properties, such as:

```
Theorem select_correct : ∀ tbl tbl' wh, select wh tbl = tbl' →
    (∀ tpl, wh tpl = true → (In tpl tbl' ↔ In tpl tbl)).
```

Our store loads grade data when the application starts, and saves grade data when the application stops. Starting and stopping is straightforward, so we do not discuss them in more detail. Correctness of persistence is reflected in the store interface by the requirement that serialization and deserialization are inverses:

```
Parameter serial    : Table n → string.
Parameter deserial  : string  → option (Table n).
Parameter serial_deserial : ∀ (tbl : Table n),
  deserial (serial tbl) = Some tbl.

Parameter serialize : ∀ (r : t) (m : [Table n]),
  Cmd (m ~~ rep r m)
      (fun str : string ⇒ m ~~ rep r m * [str = serial m]).

Parameter deserialize : ∀ (r : t) (s : string),
  Cmd (rep r nil)
      (fun m : option [Table n] ⇒
         match m with
         | None ⇒        rep r nil * [None   = deserial s]
         | Some m ⇒ m ~~ rep r m * [Some m = deserial s]
         end).
```

We have implemented the store using an abstract list which has several possible implementations, including a C-style linked-list. The higher-order nature of Ynot makes it easy to express our store operations using the list's effectful fold operation.

4.3. Verified Implementation

Based on the specification given in Section 4.1, a verified implementation of our gradebook meets the following interface:

```
Module Type GradeBookAppImpl.
  Parameter T : Set.
  Parameter rep : T → (Config * list (ID * (list Grade))) → hprop.

  Parameter exec : ∀ (t : T) (cmd : Request)
    (m : [Config * list (ID * (list Grade))]),
    Cmd (m ~~ rep t m * [gb_inv (snd m) (fst m) = true])
        (fun r : Response ⇒ m ~~ [r = fst (mutate cmd m)] *
             rep t (snd (mutate cmd m)) *
             [gb_inv (snd m) (fst m) = true]).
End GradeBookAppImpl.
```

Note that the type guarantees that any implementation of `exec` is invariant preserving and faithfully models `mutate`. For convenience, we keep the course configuration in memory at runtime, and parameterize our implementation by an abstract backend store:

```
Module GradeBookAppStoreImpl (s : Store) : GradeBookAppImpl.
  Definition T := (Config * s.T).
  (** ... **)
End GradeBookStoreImpl.
```

In trying to write `rep`, we immediately encounter an impedance mismatch between our logical gradebook model (based on `list (ID * list Grade)`) and the table based model of the store (based on `Tuples`). Following the 3-tier web application model, we define an object-relational mapping [23] between the domain-specific objects of students, grades, etc., and the relational store:

```
 1 Module GradesTableMapping.
 2   Fixpoint Tuple2List' n : Tuple n → list Grade :=
 3     match n as n return Tuple n → list Grade with
 4     | 0 ⇒ fun _ ⇒ nil
 5     | S n ⇒ fun x ⇒ (fst x) :: (Tuple2List' n (snd x))
 6     end.
 7
 8   Definition Tuple2List n (x : Tuple (S n)) :=
 9     match x with
10     | (id, gr) ⇒ (id, Tuple2List' n gr)
11     end.
12
13   Fixpoint Table2List n (x : Table (S n)) : list (ID * list Grade) :=
14     match x with
15     | nil ⇒ nil
16     | a :: b ⇒ Tuple2List n a :: Table2List n b
17     end.
18 End GradesTableMapping.
```

The **return** annotation on line 3 expresses the relationship between n and the resulting type of the **match** and is necessary for helping the Coq type checker. Other data models, such as with three-tuples (id, assignment, grade), require different mappings, but, regardless of the choice of data model and mapping, we must prove that the mapping is an isomorphism from the logical model to the data model:

```
1 Theorem TblMTbl_id : ∀ l c, store_inv1 l c = true →
2   Table2List _ (List2Table l (length (totals c))) = l.
```

Isomorphism is actually an overly strong requirement, but it helps simplify reasoning. The `store_inv1` predicate on line 1 captures when the isomorphism applies (e.g., when the length of the store's list of tuples is the same as the number of assignments).

With the mapping to the data model done, we can define the concrete imperative representation:

```
Definition rep (cfg, t) (cfg', gb) :=
  [cfg = cfg'] * s.rep t (List2Table gb)
```

The imperative implementation consists of a runtime configuration `cfg` and a handle to an imperative store `t`, which `rep` relates to the logical gradebook model. The `rep` predicate states that the runtime configuration (`cfg`) is identical to the logical model's configuration (`cfg'`), and that the imperative gradebook's state (`t`) is related to the logical model (`List2Table gb`) by the isomorphism we defined previously. The complete imperative implementation consists of hundreds of lines of code, proofs, and tactics, so we can only give highlights here. The implementation of retrieving a grade, omitting some definitions, is:

```
1  Definition F_get user pass id assign m t :
2    Cmd (m ~~ rep t m * [store_inv (snd m) (fst m) = true] *
3           [private (fst t) (GetGrade user pass id assign) = true])
4        (fun r : Response ⇒ m ~~ [store_inv (snd m) (fst m) = true] *
5           [r = fst (mutate (GetGrade user pass id assign) m)] *
6           rep t (snd (mutate (GetGrade user pass id assign) m))).
7   refine (fun user pass id assign m t ⇒
8        res ← s.select (snd t) (get_query id (fst t))
9                   (m ~~~ List2Table (snd m)
10                  (length (totals (fst t))) ) <@> _ ;
11       match nthget assign res as R
12           return nthget assign res = R → _ with
13       | None ⇒ fun pf ⇒ {{ !!! }}
14       | Some w ⇒ fun pf ⇒
15         match w as w' return w = w' → _ with
16         | None ⇒ fun pf2 ⇒ {{ Return ERR_BADGRADE }}
17         | Some w' ⇒ fun pf2 ⇒ {{ Return (RET w') }}
18         end (refl_equal _)
19       end (refl_equal _) );
20 (** Proof. **)
21   rsep fail auto.
```

The intuition here is that we first run a `get_query` over the store `s` (lines 8-10), which results in a table `res`. Because the gradebook invariant holds, `res` contains a single tuple of the requested student's grades. `nthget` returns `None` if the input table is empty, so we mark this branch as impossible (using the `!!!` command). We then project out the desired grade, returning an error if the requested assignment does not exist. The proof script for this function is almost completely automated and consists almost entirely of appeals to Ynot's built-in separation logic tactic `sep` augmented with heuristics that apply purely logical lemmas about the application model. For example, a typical proof about the specification is:

```
Theorem GetGrade_private_valid : ∀ (T : Set) (x : Config * T)
  user pass id assign,
  → store_inv (snd x) (fst x) = true
  → private (fst x) (GetGrade user pass id assign) = true
  → nthget assign (select (get_query id (fst x))
        (List2Table (snd x) (length (totals (fst x))))) <> None.
```

This theorem states that if the get request is privacy respecting:

```
private (fst x) (GetGrade user pass id assign) = true
```

and the invariant holds on the store:

```
store_inv (snd x) (fst x) = true
```

then the student has a grade:

```
nthget assign (select (get_query id (fst x))
        (List2Table (snd x) (length (totals (fst x))))) <> None
```

The other grade-manipulating operations are implemented analogously.

4.4. Deploying to an Application Server

To deploy our application using a read-parse-execute-pretty-print application server we have developed we must wrap our implementation in an `App` module:

```
Module Type App.
  Parameter Q : Set.              (** type of app's input *)
  Parameter R : Set.              (** type of app's output *)

  Parameter T : Set.              (** type of imperative app *)
  Parameter M : Set.              (** type of logical app model *)
  Parameter rep : T → M → hprop. (** representation invariant *)

  (** the functional model of the application *)
  Parameter func : Q → M → (R ∗ M).
  Parameter appIO : Q → M → (R ∗ M) → Trace.

  (** the app implementation *)
  Parameter exec : ∀ (t : T) (q : Q) (m : [M]) (tr : [Trace]),
    Cmd (tr ~~ m ~~ rep t m ∗ traced tr)
        (fun r : R ⇒ tr ~~ m ~~ let m' := snd (func q m) in
              [r = fst (func q m)] ∗
              rep t m' ∗ traced (appIO q m (r,m') ++ tr)).
```

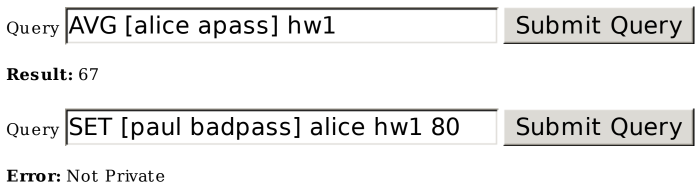
This interface requires a functional application model (`func`), and allows the application to transparently perform I/O operations by wrapping the desired sequence in the `appIO` trace. The gradebook application only performs I/O on startup and shutdown, and so it meets this interface trivially. (Of course, the application server itself performs much more I/O). The application server also requires a parser and frontend that are defined by the following functions and are discussed in the following subsection:

```
  Parameter grammar : Grammar Q.
  Parameter parser : parser_t grammar.
  Parameter printer : R → string.
  Parameter err : parse_err_t → string.
End App.
```

With these definitions in place, we can describe the traces of a correct application implementation, which we do using an inductive data type in the same way we specified correctness for the counting echo server (Section 3.5). Intuitively, either the input request parsed correctly, and the result was sent to the application for processing and the response returned to the user, or the parse failed and an error was returned; see Figure 4.

4.5. Frontend

The frontend parses inputs into `Request`s and converts application `Response`s into text. For example, we have implemented a raw-sockets frontend by straightforwardly parsing `Request`s and printing `Response`s. We have also implemented an HTML frontend as an application server (transformer). Given an application, the HTML application server passes along certain HTTP fields to the application and converts response to HTML output. Several screen shots of the web application running with a minimal skin are given in Figure 7. Here we use strings for user names, passwords and assignments which the backend supports through a lookup table that we omitted in our presentation.



AVG [alice apass] hw1

Result: 67

SET [paul badpass] alice hw1 80

Error: Not Private

Fig. 7. Screenshots of the gradebook running in Mozilla Firefox.

The HTTP server uses Ynot's packrat PEG parser toolkit [10] to parse HTTP requests. The parser is implemented as a verified compiler [31]: given a specification consisting of a PEG grammar and semantic actions, the parser creates an imperative computation that, when run over an arbitrary imperative character stream, returns a result that agrees with the specification. To make the parsing efficient, the generated parser uses the packrat algorithm which implements a sophisticated caching strategy based on imperative hash tables to make the parser efficient. By structuring this module as a compiler, we can use it to write custom parsers for a wide variety of tasks.

4.6. Verification Overhead

Figure 8 describes the breakdown of proofs, specifications, and imperative code in our verified components. Program code is Haskell-ish code that has a direct analog in the executed program (e.g. `F_get`). Specs are model definitions but not proofs (e.g. `gb_inv`). Proofs counts all proofs (e.g. `select_just`) and tactic definitions. Overhead gives the ratio of proofs to program code and the time column indicates proof search and checking time on a 2Ghz Core 2 laptop with 2GB RAM. We have made no attempt to optimize any of these numbers. These totals do not include the base Ynot tactics and data structures that we use, which include an imperative hash table, stream, and segmented linked list.

The ratios of overhead vary, but the application stands out as having the largest proof burden. This is primarily because we opted to directly reason about sets as permutation-equivalence classes of ordered lists which have no duplicate elements, instead of using a set library like [15]. As a result, details of our set implementation have complicated our proofs. We found that in general, Ynot's separation logic tactics were able to successfully isolate reasoning about the heap, reducing the problem of verification to a straightforward but non-trivial Coq programming task. For a more detailed discussion of engineering proofs with Ynot, see [10].

	Program	Specs	Proofs	Overhead	Time (m:s)
Packrat PEG Parser	269	184	82	.3	0:55
Store	113	154	99	.88	0:23
Gradebook Application	119	231	564	4.74	0:32
Application Server	223	414	231	1.04	1:21
Other I/O Library	90	76	90	1	0:05

Fig. 8. Numbers of lines of different kinds of code in the imperative components

5. Extraction

Our axiomatic treatment of imperative commands facilitates reasoning but does not allow us to actually run programs. In order to produce executable code, we must translate our Coq code into a runnable language in which we can realize our imperative axioms. Coq provides such an *extraction* mechanism [26] for translating Coq code into OCaml, Haskell, and Scheme; in our work we chose OCaml.

Coq extraction consumes well-typed Coq terms and produces OCaml terms. Because the Coq type system is so much more expressive than the OCaml type system, extraction makes heavy use of unsafe casting, written `Obj.magic`. However, because we know that our Coq program is well-typed, and hence "cannot go wrong", we know that all of these casts are actually safe.

Extraction produces OCaml code with calls to undefined functions that correspond to our axioms, e.g. `SepRead`, `SepWrite`, etc.. Therefore, in order to run our program, we must implement these functions. The Ynot library declares 11 axioms for reasoning in the `Cmd` monad. These are realized in OCaml and reused for all Ynot developments that extract to OCaml. In OCaml, type variables always have primes and type application places arguments before functions, so the type `'a F` indicates the type constructor `F` applied to type variable `'a`:

```
type 'a axiom_ST = unit → 'a

let axiom_STBind b k () = let v = b () in k v ()
let axiom_STReturn v () = v

let axiom_STContra () = failwith "ST Contradiction"

let axiom_STWeaken x = x
let axiom_STStrengthen x = x

(** ((a → (unit → b)) → a → (unit → b)) → a → (unit → b) **)
let axiom_STFix f =
  let rec fix a () = f fix a ()
  in fix

type axiom_ptr = Obj.t ref

let axiom_STNew v () = ref (Obj.magic v)
let axiom_STFree p () = ()
let axiom_STRead p () = Obj.obj !p
let axiom_STWrite p v () = (p := Obj.repr v)
```

These axioms are defined at the non-separation logic (raw heap) level, and the corresponding lower-level monad is called `ST` instead of `Cmd`. Ynot implements the `Cmd` monad on top of the `ST` monad in Coq. Our Coq computations correspond to OCaml *thunks*, which are effectful computations suspended inside of lambdas. The `bind` and `return` operations have standard definitions. Contradiction is implemented by raising a runtime exception, but this operation can never be called because the Coq axiom can only be used in provably dead code. `Weaken` and `Strengthen` correspond to strengthening preconditions and weakening post-conditions, and they are no-ops because they are only used to reason about correctness. The general-recursive `Fix` combinator has the standard definition for a call-by-value language. Finally, we implement the heap operations using OCaml references. Since Ynot pointers are untyped and support strong-update, we model `ptr` as a reference to `Obj.t` which is OCaml's equivalent to C's `void*`. `New` returns a new reference and free is a no-op since OCaml is garbage collected. Finally, `Read` and `Write` have standard definitions based on reading and writing references in OCaml.

In addition to the Ynot core, our axiomatization of traces also requires an OCaml realization. We need to give computational definitions for axioms involving files, server sockets, and UDP, TCP, and SSL sockets. In total, this basis includes 23 definitions that are mostly modularly defined on top of our file abstraction. The file abstraction provides basic operations, like `read`, `write`, `flush`, and `close`, and are implemented for each type of file and persistent socket. The implementation for regular Unix files is:

```
(** File Interface **)
type file_descriptor = {
    fd    : Unix. file_descr ;
    read  : unit → char option;
    write : char → unit → unit;
    flush : unit → unit;
    close : unit → unit
  }

let axiom_read _ _ fd () =
  match fd.read () with
    None   → None
  | Some r → Some (ctoa r)
let axiom_write _ _ fd chr () = fd.write (atoc chr) ()
let axiom_flush _ _ fd = fd.flush
let axiom_close _ _ fd = fd.close

(** File Implementation **)
let file_read fd () =
  let str = String.create 1 in
  let len = Unix.read fd str 0 1 in
  if len = 1 then Some (String.get str 0)
  else None
let file_write fd c () =
  let str = String.make 1 c in
  let len = Unix.write fd str 0 1 in
  assert (len = 1);
  ()
let file_close fd () = Unix.close fd
let file_flush fd () = flush (Unix.out_channel_of_descr fd)
```

Having realized all of the required axioms, we can compile our code with the OCaml compiler. The generated code derives its correctness from the Coq type checker, the Coq extraction mechanism, the realization of the computational axioms just presented, the correctness of the OCaml compiler and libraries (including the system calls that they use), and the soundness of our Ynot extensions. While this is still a considerable amount of code to trust, it is also possible to verify these lower-level components [14, 4]. The soundness of the Ynot axioms is discussed in [39].

6. Related Work

6.1. Weaker notions of correctness

Our approach to building verified web systems is to prove them correct by construction at development time. Alternatively, pre-existing applications can be verified to be free of certain errors through static analysis. In [19], for example, the authors rule out SQL injection attacks for a large fragment of PHP using an information flow analysis to ensure that tainted application inputs are never used in SQL queries. Their notion of correctness is the absence of certain classes of errors; with Ynot we can prove correctness with respect to an arbitrary logical model of application behavior, which may itself specify the absence of injection attacks. And although we have specified our logical gradebook model in Coq, specifications can be developed using special-purpose tools such as Alloy [20]. Moreover, in Ynot, reasoning is modular and compositional: interfaces themselves guarantee correctness properties; in [19], the entire program must be analyzed. See [9] for a discussion of how a wide range of correctness properties can be obtained with minimal effort using a dependent type theory such as Coq.

6.2. Alternative Approaches to Full Verification

Interest in the full verification of higher-order imperative software has surged recently [25], and in this section we highlight several alternative verification methodologies and other verified software case studies.

Jahob [44] is similar to Ynot. It allows users to write effectful Java code, which is automatically verified against a programmer specified logical model by a combination of automated theorem provers. Although Jahob is also based on a Hoare logic, it does not use separation logic for reasoning about memory and requires a significantly larger trusted code base than Ynot. Moreover, Jahob cannot be used to reason about I/O behavior. To the best of our knowledge, Jahob has never been used to certify a system like ours.

The Isabelle/HOL [36] proof assistant can be used to verify higher-order imperative programs. In [8], the authors verify an array-based checker for resolution proofs and a bytecode verifier using an approach similar to our own. However, our work differs in several key respects. First, our work in based on Hoare type theory, so the types of imperative computations specify their behavior. Because Isabelle/HOL lacks dependent types, it is difficult to parameterize a computation monad by pre- and post-conditions, and as a result, the authors must use an explicit operational semantics to reason about imperative computations "after the fact". For example, using Ynot notation, the type of a pointer-swapping function in [8] would be:

```
swap : ptr → ptr → Cmd unit
```

Many of the correctness properties that come for free in Ynot, such as the prevention of array indexing out of bounds, must be explicitly proved (without built-in support for separation logic) using Isabelle/HOL. As such, if we are interested in these properties, more effort is required to establish them, but if we are not interested in these properties, then in Isabelle/HOL we are not obligated to prove them. Another key difference between [8] and our work is the expressivity of the underlying language and logic: the programming language of [8] is restricted to storing first-order values in the heap and its logic cannot be used to reason about I/O behavior. Ynot allows higher-order values and imperative computations to be stored in the heap; see [42] for an approach that allows higher-order values, but not computations, to be stored in the heap.

Isabelle/HOL has been used to verify a number of software systems. In [41], the authors verify a concurrent OCaml implementation of a distributed queue running the alternating bit protocol. Their development is similar in spirit to ours, but differs in scale - our development is 800 lines of code and 2000 lines of specification and proof, and the queue is 3000 lines of code and 3000 lines of specification and proof. Our development can be verified in several minutes; the queue takes an hour. Isabelle/HOL has also been used to verify the seL4 microkernel [24], but the verification overhead is on the order of 20x, with a 200,000 line proof of correctness. A distinguishing feature of Ynot is its focus on lightweight proof automation, for separation logic in particular, which is discussed in [10].

ACL2 [22] is a first-order applicative language and proof assistant used in a variety of industrial settings, often to verify hardware. Unlike Coq, ACL2 does not output separately checkable proof terms (and hence has a larger trusted base than Coq). An extension of ACL2, dubbed single-threaded objects [6], allows one to write a restricted class of imperative programs and reason about them as though they were purely functional. This allows for large speed ups when, e.g, lists may be destructively modified in place. Indeed, the motivation for the single-threaded objects extension is to increase the speed of the prover, rather than allow users to program in an unrestricted imperative style. Programs that use imperative state must obey syntactic restrictions that guarantee that there is only one reference to any stateful object. It is unclear if this methodology can be pushed to a setting as general as ours, but an extension to Ynot that allows for "safe" computations (e.g., factorial) to be executed during type-checking would be useful.

6.3. Program Derivation from Specifications

Declarative networking [29] is an approach to building distributed systems by generating them from protocol descriptions written in domain specific languages based on Prolog. Realistic protocols written in this style are often remarkably concise, and they can also be reasoned about using techniques from automated theorem proving [43]. Our extended Ynot is a natural language choice for the code generated from such a system, as we could potentially prove that properties that hold at the declarative protocol level continue to hold in the generated code. See [43] for references on other approaches, besides theorem proving, to prove properties of declarative networking specifications. A similar line of work aims to extract concurrent programs from provable propositions in an extensional type theory [3].

7. Conclusion

We learned a number of lessons in building our verified gradebook server. The first is the importance of the logical specification of application behavior. An imperative algorithm will be difficult to verify if its functional model is difficult to reason about. Although we have sketched the specification of our gradebook server in this paper, the properties that our implementation guarantees can only truly be understood by examining the Coq gradebook specification. Such guarantees also depend on lower level specifications. For example, our networking library does not capture timeout, retry, or filesystem behavior, making certain properties difficult or impossible to specify without modifying the I/O library. And because Hoare logic only captures partial correctness, the divergent computation is a verified implementation of every specification.

One possible future direction is to further refine the I/O library to take additional behaviors into account. Another direction is to integrate the gradebook with a more realistic relational database [17]. Finally, our server is single threaded but concurrency can be added to separation logic [7] and transactions can also be considered [32].

References

[1] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: the spi calculus. In *CCS '97: Proceedings of the 4th ACM conference on Computer and communications security*, pages 36–47, New York, NY, USA, 1997. ACM.

[2] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

[3] Mark Bickford and Robert L. Constable. A logic of events. Technical report, Cornell University, 2003.

[4] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: Hol specification and symbolic-evaluation testing for tcp implementations. In *Priciples of Programming Languages '06*, pages 55–66, New York, NY, USA, 2006. ACM.

[5] Bard Bloom, Sorin Istrail, and Albert R. Meyer. Bisimulation can't be traced. *Journal of the ACM*, 42(1):232–268, 1995.

[6] Robert S. Boyer and J. Strother Moore. Single-threaded objects in acl2. In *Practical Aspects of Declarative Languages (PADL), volume 2257 of LNCS*, pages 9–27. Springer-Verlag, 1999.

[7] Stephen Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1-3):227–270, 2007.

[8] Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkök, and John Matthews. Imperative functional programming with isabelle/hol. In *TPHOLs '08: Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, pages 134–149, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] Adam Chlipala. Ur: statically-typed metaprogramming with type-level record computation. *SIGPLAN Not.*, 45(6):122–133, 2010.

[10] Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *Proceedings of the 14th ACM SIGPLAN International Confernence on Functional Programming*, 2009.

[11] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[12] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Unraveling the web services web: An introduction to soap, wsdl, and uddi. *IEEE Internet Computing*, 6(2):86–93, 2002.

[13] David Delahaye. A tactic language for the system Coq. In *Proceedings of the International Conference on Logic Programming, Artificial Intelligence, and Reasoning*, 2000.

[14] Xinyu Feng, Zhong Shao, Yuan Dong, and Yu Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 170–182, New York, NY, USA, 2008. ACM.

[15] Jean-Christophe Filliâtre and Pierre Letouzey. Functors for proofs and programs. In *European Symposium on Programming 2004*, pages 370–384, 2004.

[16] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *POPL '04: Proceedings of the 31st annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 111–122, New York, NY, USA, 2004. ACM.

[17] Gregory Malecha and Ryan Wisnesky and Avraham Shinnar and Greg Morrisett. Certified Data Management in Ynot. In *POPL'10: Principles of Programming Languages*. ACM, January 2009.

[18] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *SIGPLAN Notices*, 43(1):273–284, 2008.

[19] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *WWW '04*, pages 40–52, New York, NY, USA, 2004. ACM.

[20] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 2002.

[21] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *POPL '93: Proceedings of the 20th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 71–84, New York, NY, USA, 1993. ACM.

[22] Matt Kaufmann and J S. Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Transactions on Software Engineering*, 23:203–213, 1997.

[23] Wolfgang Keller. Mapping objects to tables: A pattern language. In *Proceedings Of European Conference on Pattern Languages of Programming Conference (EuroPLOP) '97*, 1997.

[24] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, New York, NY, USA, 2009. ACM.

[25] K. Rustan M. Leino. Learning to do program verification. *Commun. ACM*, 53(6):106–106, 2010.

[26] P. Letouzey. A new extraction for Coq. *Lecture Notes in Computer Science*, 2646:200–219, 2003.

[27] Andres Lőh and Ralf Hinze. Open data types and open functions. In *International Conference on Principles and Practice of Declarative Programming*, pages 133–144, New York, NY, USA, 2006. ACM Press.

[28] Barbara Liskov and Stephen N. Zilles. Specification techniques for data abstractions. *IEEE Transactions of Software Engineering*, 1(1):7–19, 1975.

[29] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking: language, execution and optimization. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 97–108, New York, NY, USA, 2006. ACM.

[30] Conor McBride and James McKinna. The view from the left. *Journal Functional Programming*, 14(1):69–111, 2004.

[31] James Mckinna and Joel Wright. A type-correct, stack-safe, provably correct expression compiler in epigram. In *Journal of Functional Programming*, 2006.

[32] Aleksandar Nanevski, Paul Govereau, and Greg Morrisett. Towards type-theoretic semantics for transactional concurrency. In *The 4th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 79–90, New York, NY, USA, 2008. ACM.

[33] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in hoare type theory. *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, 2006.

[34] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: dependent types for imperative programs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, 2008.

[35] Gleb Naumovich and Paolina Centonze. Static analysis of role-based access control in j2ee applications. *SIGSOFT Software Engineering Notes*, 29(5):1–10, 2004.

[36] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic.* Springer-Verlag, London, UK, 2002.

[37] Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Chalmers University of Technology, 2007.

[38] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL '01: Proceedings of the 10th Annual Conference of the European Association for Computer Science Logic*, 2001.

[39] Rasmus Lerchedahl Petersen, Lars Birkedal, Aleksandar Nanevski, and Greg Morrisett. A realizability model for impredicative hoare type theory. In Sophia Drossopoulou, editor, *European Symposium on Programming 2008*, volume 4960 of *Lecture Notes in Computer Science*, pages 337–352. Springer, 2008.

[40] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *IEEE Symposium on Logic in Computer Science*, 2002.

[41] Thomas Ridge. Verifying distributed systems: the operational approach. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 429–440, New York, NY, USA, 2009. ACM.

[42] Wouter Swierstra. *A functional specification of effects*. PhD thesis, University of Nottingham, 2009.

[43] Anduo Wang, Prithwish Basu, Boon Thau Loo, and Oleg Sokolsky. Declarative network verification. In *PADL '09: Proceedings of the 11th International Symposium on Practical Aspects of Declarative Languages*, pages 61–75, Berlin, Heidelberg, 2009. Springer-Verlag.

[44] Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. In *PLDI '08: Proceedings of the Conference of Programming Language Design and Implementation*, 2008.