Certified Web Services in Ynot

Ryan Wisnesky, Gregory Malecha, and Greg Morrisett

Harvard University
{ryan, gmalecha, greg}@cs.harvard.edu

Abstract. In this paper we demonstrate that it is possible to implement certified web systems in a way not much different from writing Standard ML or Haskell code, including use of imperative features like pointers, files, and socket I/O. We present a web-based course gradebook application developed with Ynot, a Coq library for certified 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 behavior. Expressive abstractions allow the modular certification of both high level specifications like privacy guarantees and low level properties like memory safety and correct parsing.

1 Introduction

In this paper we demonstrate that it is possible to implement certified web systems in a way not much different from writing Standard ML or Haskell code, including use of imperative features like pointers, files, and socket I/O. We present a web-based course gradebook application developed with Ynot [18], a Coq [1] library for certified, general-purpose, imperative programming.

Our specification of application behavior, imperative application implementation, and certification that the implementation meets the specification are all written in the dependently-typed, higher-order λ-calculus of inductive constructions (CIC) [1]. Ynot links user code with an imperative extension to the Coq proof assistant, resulting in an executable web server. We add a dialog-based [12] I/O system to Ynot, and we extend Ynot's underlying Hoare logic with event traces [3] to reason about I/O behavior. Expressive abstractions allow the modular certification of both high level specifications like privacy guarantees and low level properties like memory safety and correct parsing. The proof that the application meets its specification has been developed semi-automatically and interactively during development, imposes no runtime overhead, and can be verified in minutes by a several-hundred-line CIC typechecker.

We give an overview of the course gradebook server in Section 2 and describe Ynot in Section 3. Our I/O library is described in Section 4. The certified imperative implementation is discussed in Section 5, followed by a comparison to related tools in Section 6. We conclude with lessons learned and thoughts on future work. The source code is included in the Ynot distribution at `ynot.cs.harvard.edu`.

2 Overview

Our goal is to build a simple, web-based course gradebook that allows students, teaching assistants, and professors the ability to read, edit, and statistically aggregate grades 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 [19].

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 [20]. External databases may also be used this way.

The application logic specifies the behavior of the gradebook using high-level domain-specific concepts like grades, assignments, and sections. For instance, the specification states that students should not be allowed to see 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, with commands sent to the application using HTTP. The application server parses HTTP requests by compiling input PEG grammars [8] to packrat parsing computations in a certified way [15]. An executable webserver is generated by extraction from Coq to OCaml [1].

3 Ynot

In this section we introduce Ynot by means of increasingly comprehensive examples. We begin with `helloworld`:

```
Definition helloworld : Cmd empty (fun _ : unit => empty) :=
  printStringLn "Hello World".
```

Imperative computations (also called commands) like `printStringLn` 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 [17]. Ynot allows non-terminating recursion, and post-conditions are partial correctness assertions. Hence `helloworld` is a computation whose pre-condition is that the heap is empty, and whose post-condition is that the computation, if it doesn't diverge, returns a unit and ensures that the heap is empty.

The heap is accessed through the traditional new, read, write, free commands, which we reason about using separation logic [20]. Their types are:

```
Definition SepNew (T : Type) (v : T) : Cmd empty (fun p => p --> v).

Definition SepFree (T : Type) (p : ptr)
    : Cmd (Exists v :@ T, p --> v) (fun _ : unit => empty).

(* SepRead is also written ! *)
```

```
Definition SepRead (T : Type) (p : ptr) (P : T -> hprop)
    : Cmd (Exists v :@ T, p --> v * P v) (fun v => p --> v * P v).

(* SepWrite is also written ::= *)
Definition SepWrite (T T' : Type) (p : ptr) (v : T')
    : Cmd (Exists v' :@ T, p --> v') (fun _ : unit => p --> v).
```

Pre- and post-conditions are predicates over the heap (`hprop`s). A `:@` indicates the type of an existential quantifier. The `p --> v` represents the `hprop` that `p` points to `v`. For example, when `SepNew` is run in the empty heap with argument `v`, it returns a pointer[1] 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 in to two disjoint portions that satisfy each conjunct. `SepRead`'s type 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 is useful for proof automation but not strictly required.

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 v`, and as `c1 ;; v2` 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`. In general, we write imperative code first and then prove the correctness of sequencing afterward, using a Coq tactic called `refine`. For example, the following program swaps the values of two pointers:

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

The type of `swap` expresses that `swap` takes as arguments two pointers and two computationally irrelevant natural numbers such that the first pointer points to the value of the first and analogously for the second. If `swap` terminates, then the first pointer will point to the second value and the second will point to the first value. The `[-]` marks a parameter as computationally irrelevant. Such parameters only serve specification purposes and do not affect runtime

[1] Ynot does not allow unrestricted pointer arithmetic, so pointers are essentially references/locations.

behavior; they are erased during compilation. Irrelevant values must be explicitly unpacked: `v1` has type `[nat]`, but inside the `v1`$\sim\sim$ it can be treated as a `nat`.

The `swap` function itself is similar to a typical pointer-swapping function but includes extra information to help us prove partial correctness. `refine` generates proof obligations, which we here discharge using Ynot's built in separation logic tactic, `sep`. Within the `refine`, `<@>` is a use of separation logic's frame rule, which allows us to describe the portion of the heap that a computation doesn't use. In this example, for instance, we need to know that `p2` points to `v2` before and after `p1` is read. This fact can actually be inferred automatically, but we write it out here for sake of explanation. In the following line, the `_` indicates that the framing condition should be inferred. Finally, the {{-}} indicates that the type of the final write may need its pre-condition strengthened and its post-condition weakened to match the overall type of `swap`.

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

```
Definition leak : Cmd empty (fun _ : unit => empty).
  refine (v <- SepNew 1 ; {{ Return tt }}).
```

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 a false obligation:

```
v --> 1 ==> empty
```

We achieve modularity in Ynot by defining abstract interfaces for imperative components so that many implementations can be used. Consider the following interface and implementation of a simple counter:

```
Module Type Counter.
  Parameter T : Type.  (* type of implementation *)
  Definition M := nat. (* type of logical model  *)
  Parameter rep : T -> M -> hprop. (* heap representation *)

  Parameter inc : forall (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.
```

`T` is the type of the imperative implementation, which corresponds to a pointer for this implementation. `M` is the logical model for the data structure, in this case a natural number which is the current value in the counter. The `rep` function relates, through an `hprop`, the state of the imperative implementation to logical model. The `forall` keyword indicates a dependent function type: `inc`'s post-condition depends on `m`. The module type hides everything but the logical model, providing an abstraction barrier for users of the module.

4 Files, Sockets, and Traces

We have extended Ynot with an axiomatic networking and file I/O library based on the OCaml Unix library. Just as we record the effect of memory operations using separation logic, we record the effects of I/O actions using a trace [3]:

```
Axiom Action : Set.
Definition Trace := list Action.
Axiom traced : Trace -> hprop.
```

Our type of `Action`s is open [14], allowing library users to define additional I/O events. Traces are defined as lists for convenience, and we will only be reasoning about finite trace fragments.

We include file and socket operations such as read, write, accept, etc.. The UDP send operation, for instance, is exposed as (List cons is written as `::` in Coq):

```
Axiom SockAddr : Set.
Axiom Sent : SockAddr -> SockAddr -> list ascii -> Action.

Axiom send : forall (local remote : SockAddr)
 (s : list ascii) (tr : [Trace]),
   Cmd (tr ~~ traced tr)
       (fun _ : unit => tr ~~ traced (Sent local remote s :: tr)).
```

We do not formally verify our OCaml code, which for the most part delegates to Unix functions. For instance, we do not verify the implementation of the TCP state machine, although it is possible to do so [2].

Traces can be reasoned about using temporal logics [6]; however, for simplicity, we reason about them directly using inductive Coq definitions. For instance, the following Coq datatype specifies correct, properly echoed, traces of an echo server:

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

Here each `|` indicates a data constructor. 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`). The `|` symbol is also used in `match` expressions that eliminate inductive types.

4.1 Certified 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 [12], and our I/O library contains support for proving such systems correct with respect to a trace [3]

semantics. At a minimum, an application iteration is defined by an invariant-preserving Ynot function that is runnable in the initial world of an empty heap and empty trace. For instance, the type of an echo application is:

```
Definition echo_iter_t local := forall (tr : [Trace]),
 Cmd (tr ~~ traced tr * [echoes local tr])
       (fun _ : unit => tr ~~ Exists r :@ Trace,
          traced (r ++ tr) * [echoes local (r ++ tr)]).
```

The `[]` notation is overloaded here to indicate "pure" propositions which do not mention the heap. List concatenation is written `++`. A computation of this type, when repeated any number of times, beginning in the initial world, always generates a trace that is in `echoes`. An example echo implementation that conforms to the above specification is:

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

We have written out the intermediate state of the trace history (using an irrelevant value unpacking operation $\sim\sim\sim$), but such states can often be inferred.

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.

5 The Application

In this section we describe the gradebook application specification, our imperative implementation of it, and the proof that the implementation meets the specification. We begin with the purely functional specification of the gradebook itself (Section 5.1). We then describe the entire deployed application server starting from the backend and working toward the user. We start with the data store (Section 5.2) which provides the data manipulation operations we use in our imperative implementation (Section 5.3). From there, we show how the application can be deployed using our application server (Section 5.4). We conclude with an explanation of the frontend (Section 5.5) in which we focus on parsing user requests. It is helpful to keep in mind that every imperative component must be related to a purely functional model.

5.1 Application Logic

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

```
Definition Section    := nat.
Definition Password   := nat.
Definition Grade      := nat.
Definition Assignment := nat.
Record Config : Set := mkCfg {
  students, tas, profs : list ID;
  sections : list (ID * Section );
  hashes   : list (ID * Password);
  totals   : 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. These properties are given by a Coq definition:

```
Definition correct_cfg (cfg : Config) := forall id,
    (In id (students cfg) \/ In id (tas cfg) \/ In id (profs cfg) ->
      exists hash, lookup id (hashes cfg) = Some hash) /\ ...
```

The actual grades are modeled by a `list (ID * list Grade)`. Like 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 totals list given in the configuration, each grade must be less than the associated maximum permissible, and each student must have an entry.

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

```
Inductive Command : Set :=
| SetGrade : ID -> PassHash -> ID -> Assignment -> Grade -> Command
| GetGrade : ID -> PassHash -> ID -> Assignment -> Command
| Average  : ID -> PassHash -> Assignment -> Command.
```

The meaning of the commands is given by a pure Coq function `mutate` that maps a `Command`, `Config`, and `list (ID * list Grade)` to a new `list (ID * list Grade)` and a response:

```
Inductive Response : Set :=
   | ERR_NOTPRIVATE : Response | ERR_BADGRADE : Response
   | ERR_NOINV : Response | OK  : Response | RET : Grade -> Response.
```

There are numerous ways to define the desired gradebook behavior, but using a total function makes the application easy to deploy with our application server.

Privacy is enforced using simple role based access control; `private` is a predicate that defines when commands are privacy respecting, and non-private queries are specified to return `ERR_NOTPRIVATE`:

```
Definition isProf (cfg: Config) (id: ID) (pass: Password) :=
 In id (profs cfg)  /\ lookup id (hashes cfg) = Some pass.
...

Definition private (cfg : Config) (cmd : Command) : 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.
```

We have proved a number of theorems about this specification, like that `mutate` preserves `gb_inv` and will 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.

5.2 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 datatype:

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

For simplicity we are storing only natural numbers, and we specify only a small subset of the functionality of SQL, including select, update, project, and delete. For instance, 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, like:

```
Theorem select_just : forall tbl tbl' wh, select wh tbl = tbl' ->
    (forall tpl, In tpl tbl' -> wh tpl = true /\ In tpl tbl).

Theorem select_all : forall tbl tbl' wh, select wh tbl = tbl' ->
     (forall tpl, In tpl tbl -> wh tpl = true -> In tpl tbl').
```

Persistence is reflected in the store interface by the simple requirement that serialization and deserialization be inverses:

```
Parameter serial   : Table n -> string.
Parameter deserial : string  -> option (Table n).
Parameter serial_deserial : forall (tbl : Table n),
  deserial (serial tbl) = Some tbl.

Parameter serialize : forall (r : t) (m : [Table n]),
Cmd (m ~~ rep r m) (fun str:string => m ~~ rep r m * [str = serial m]).
Parameter deserialize : forall (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 linked-list. The linked-list has a several imperative implementations, including one using pointers to list segments. We found the linked-list's effectful fold operation particularly useful.

5.3 Certified Implementation

Based on the specification given in Section 5.1, a certified 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 : forall (t : T) (cmd : Command)
   (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 `exec` computation is invariant preserving, can only be run when the invariant is satisfied, and faithfully models `mutate`. For convenience, we keep the course configuration in memory at runtime, and we parameterize our implementation by an abstract backend store:

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

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 [13] between the domain-specific objects of students, grades, etc., and the relational store:

```
Module GradesTableMapping.
  Fixpoint Tuple2List' n : Tuple n -> list Grade :=
    match n as n return Tuple n -> list Grade with
      | 0 => fun _ => nil
      | S n => fun x => (fst x) :: (Tuple2List' n (snd x))
    end.
  Definition Tuple2List n (x : Tuple (S n)) :=
    match x with
      | (id, gr) => (id, Tuple2List' n gr)
    end.
  Fixpoint Table2List n (x : Table (S n)) : list (ID * list Grade) :=
   match x with
     | nil => nil
     | a :: b => Tuple2List n a :: Table2List n b
   end.
End GradesTableMapping.
```

The list to table direction is similar. 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:

```
Theorem TblMTbl_id : forall l c, store_inv1 l c = true ->
  Table2List (length (totals c))
   (List2Table l (length (totals c))) = l.
```

Isomorphism is actually an overly strong requirement, but it helps simplify reasoning. 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. `rep` states that the runtime configuration (`cfg`) is identical to the logical model's configuration (`cfg'`), and that the imperative gradebook's state (`t`) is isomorphic to the logical model's (`List2Table gb`). 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:

```
  Definition F_get user pass id assign m t :
    Cmd (m ~~ rep t m * [store_inv (snd m) (fst m) = true] *
               [private (fst t) (GetGrade user pass id assign) = true])
          (fun r : Response => m ~~ [store_inv (snd m) (fst m) = true] *
            [r = fst (mutate (GetGrade user pass id assign) m)] *
            rep t (snd (mutate (GetGrade user pass id assign) m))).
  refine (fun user pass id assign m t =>
```

```
res <- s.select (snd t) (get_query id (fst t))
           (m ~~~ List2Table (snd m)
           (length (totals (fst t))) ) <@> _ ;
match nthget assign res as R
 return nthget assign res = R  -> _  with
  | None => fun pf => {{ !!! }}
  | Some w => fun pf =>
  match w as w' return w = w' -> _ with
    | None =>    fun pf2 =>  {{ Return ERR_BADGRADE }}
    | Some w' => fun pf2 =>
      {{ Return (RET w')
         <@> (m ~~ [fst t = fst m] *
         [store_inv (snd m) (fst t) = true] *
         [private (fst t) (GetGrade user pass id assign) = true])}}
  end (refl_equal _)
end   (refl_equal _)  ).
```

The intuition here is that we fist run a `get_query` over the store `s`, 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 (`!!!`). We then project out the desired grade, returning an error if there is no such requested assignment. The proof script for this function is almost completely automated and consists almost entirely of appeals to `sep` and uses of purely logical theorems about the application model. For instance, a typical proof about the specification is:

```
Theorem GetGrade_private_valid : forall (T : Type) x (t : Config * T)
  user pass id assign, fst t = fst x
  -> store_inv (snd x) (fst x) = true
  -> private (fst t) (GetGrade user pass id assign) = true
  -> nthget assign (select (get_query id (fst t))
         (List2Table (snd x) (length (totals (fst t))))) <> None
```

This theorem states that if the get command is privacy respecting, then the student has a grade. The other operations are implemented analogously.

5.4 Deploying to an Application Server

To deploy our application using our read-parse-execute-prettyprint application server we must implement:

```
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 : Type.              (** type of logical app model *)
  Parameter rep : T -> M -> hprop. (** app 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 : forall (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. (Startup and shutdown are straightforward, so we do not discuss them.) The application server also requires a parser and frontend, which are defined by the following functions and discussed in the following subsection:

```
Parameter grammar : Grammar Q.
Parameter parser  : parser_t grammar.
Parameter printer : R -> list ascii.
Parameter err : parse_err_t -> list ascii.
```

With these definitions in place, we can describe the traces of a correct application implementation, which we do using an inductive datatype in the same way we specified correctness for the echo server (Section 4). 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.

5.5 Frontend

The frontend parses inputs into `Command`s and converts application `Response`s into text. For instance, we have implemented a raw-sockets frontend by using our packrat PEG parser and straightforwardly printing responses. We have also implemented an HTML frontend as an application transformer: given an application, the HTML frontend passes along certain HTTP fields to the application and converts responses to HTML output. Several screen shots of the application running with a minimal skin are given in Figure 1.

The HTTP server uses a packrat PEG parser written in Ynot to parse HTTP requests. The parser is implemented as a certified compiler [15]: 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 packrat algorithm employed by the resultant imperative computation uses a sophisticated caching strategy which is implemented using imperative hashtables. We may also write our own parsers against the parser interface.

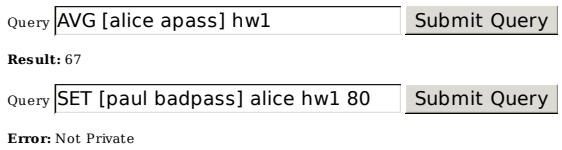

Fig. 1. Screenshots of the gradebook running in Mozilla Firefox.

5.6 Evaluation

Figure 2 describes the breakdown of proofs, specifications, and imperative code in our certified 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 hashtable, stream, and segmented linked list.

	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
HTTP-SSL-TCP Application Server	223	414	231	1.04	1:21
Other I/O Library	90	76	90	1	0:05

Fig. 2. Numbers of lines of different kinds of code in the imperative components

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 [7]. 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 succesfully isolate reasoning about the heap, reducing the problem of certification to a straightforward but non-trivial Coq programming task. For a more detailed discussion of engineering proofs with Ynot, see [5].

6 Related Work

Our approach to building certified web systems is to prove them correct by construction at development time. Alternatively, pre-existing applications can be

certified to be free of certain errors through static analysis. In [10], for instance, 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 without first being checked for validity. 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 [11]. Moreover, in Ynot, reasoning is modular: interfaces themselves guarantee correctness properties; in [10], the entire program must be analyzed. Finally, automated static checking is often unsound and prone to false positives.

Jahob [21] 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. Jahob does not use separation logic for reasoning about memory and requires a significantly larger trusted code base than Ynot. To the best of our knowledge, Jahob has never been used to certify a system like ours.

7 Conclusion

We learned a number of lessons in building our certified gradebook server. The first is the importance of the logical specification of application behavior. Even the most beautiful imperative algorithm will be difficult to certify if its functional model is difficult to reason about. And perhaps just as important is knowing what the specification does not capture. For instance, our networking library does not capture timeout and retry behavior and we do not model filesystem behavior, making certain applications difficult or impossible to specify without modifying the I/O library. Additionally because Hoare Logic only captures partial correctness, the divergent computation is a certified implementation of every specification.

Real-world web systems are considerably more complex than our gradebook application, and Ynot's feasibility at larger scales is still untested. Indeed, we are only now building executable applications (rather than only datastructures) with Ynot. But given that realistic systems will invariably require imperative features, we believe our results here are a good start.

One possible future direction is to further refine the I/O library to take additional behaviors into account. Another direction is to certifying a more realistic imperative database [9]. It is likely that the results of such an effort would also be useful in certifying realistic filesystems. Finally, our server is single threaded but concurrency can be added to separation logic [4] and transactions can also be considered [16].

References

1. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development.* Texts in Theoretical Computer Science. Springer Verlag, 2004.
2. 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 *POPL '06*, pages 55–66, New York, NY, USA, 2006. ACM.
3. Bard Bloom, Sorin Istrail, and Albert R. Meyer. Bisimulation can't be traced. *J. ACM*, 42(1):232–268, 1995.
4. Stephen Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270, 2007.
5. Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *Proc. ICFP*, 2009.
6. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.
7. Jean-Christophe Filliâtre and Pierre Letouzey. Functors for proofs and programs. In *ESOP*, pages 370–384, 2004.
8. Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *POPL '04*, pages 111–122, New York, NY, USA, 2004. ACM.
9. Carlos Gonzalia. Towards a formalisation of relational database theory in constructive type theory. In *RelMiCS*, pages 137–148, 2003.
10. 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.
11. Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.
12. Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *POPL '93*, pages 71–84, New York, NY, USA, 1993. ACM.
13. Wolfgang Keller. Mapping objects to tables: A pattern language. In *EuroPLOP*, 1997.
14. Andres Lőh and Ralf Hinze. Open data types and open functions. In *PPDP '06*, pages 133–144, New York, NY, USA, 2006. ACM Press.
15. James Mckinna and Joel Wright. A type-correct, stack-safe, provably correct expression compiler in epigram. In *JFP*, 2006.
16. Aleksandar Nanevski, Paul Govereau, and Greg Morrisett. Towards type-theoretic semantics for transactional concurrency. In *TLDI '09*, pages 79–90, New York, NY, USA, 2008. ACM.
17. Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in hoare type theory. *Proc. ICFP*, 2006.
18. Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: dependent types for imperative programs. In *Proc. ICFP*, 2008.
19. Gleb Naumovich and Paolina Centonze. Static analysis of role-based access control in j2ee applications. *SIGSOFT Softw. Eng. Notes*, 29(5):1–10, 2004.
20. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Symposium on Logic in Computer Science, LICS'02*, 2002.
21. Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. In *Proc. PLDI*, 2008.