Effective Interactive Proofs for Higher-Order Imperative Programs *

Adam Chlipala Gregory Malecha Greg Morrisett Avraham Shinnar Ryan Wisnesky

Harvard University, Cambridge, MA, USA

{adamc, gmalecha, greg, shinnar, ryan}@cs.harvard.edu

Abstract

We present a new approach for constructing and verifying higher-order, imperative programs using the Coq proof assistant. We build on the past work on the Ynot system, which is based on Hoare Type Theory. That original system was a proof of concept, where every program verification was accomplished via laborious manual proofs, with much code devoted to uninteresting low-level details. In this paper, we present a re-implementation of Ynot which makes it possible to implement fully-verified, higher-order imperative programs with reasonable proof burden. At the same time, our new system is implemented entirely in Coq source files, showcasing the versatility of that proof assistant as a platform for research on language design and verification.

Both versions of the system have been evaluated with case studies in the verification of imperative data structures, such as hash tables with higher-order iterators. The verification burden in our new system is reduced by at least an order of magnitude compared to the old system, by replacing manual proof with automation. The core of the automation is a simplification procedure for implications in higher-order separation logic, with hooks that allow programmers to add domain-specific simplification rules.

We argue for the effectiveness of our infrastructure by verifying a number of data structures and a packrat parser, and we compare to similar efforts within other projects. Compared to competing approaches to data structure verification, our system includes much less code that must be trusted; namely, about a hundred lines of Coq code defining a program logic. All of our theorems and decision procedures have or build machine-checkable correctness proofs from first principles, removing opportunities for tool bugs to create faulty verifications.

Categories and Subject Descriptors F.3.1 [*Logics and meanings of programs*]: Mechanical verification; D.2.4 [*Software Engineering*]: Correctness proofs, formal methods, reliability

General Terms Languages, Verification

Keywords functional programming, interactive proof assistants, dependent types, separation logic

* This research was supported in part by a gift from Microsoft Research and a National Science Foundation Graduate Research Fellowship.

ICFP'09, August 31–September 2, 2009, Edinburgh, Scotland, UK.

1. Introduction

A key goal of type systems is to prevent "bad states" from arising in the execution of programs. However, today's widely-used type systems lack the expressiveness needed to catch language-level errors, such as a null-pointer dereference or an out-of-bounds array index, let alone library- and application-specific errors such as removing an element from an empty queue, failing to maintain the invariants of a balanced tree, or forgetting to release a critical resource such as a database connection. For safety- and security-critical code, a type system should ideally let the programmer assign types to libraries such that client code cannot suffer from these problems, and, in the limit, the type system should make it possible for programmers to verify that their code is correct.

There are many recent attempts to extend the scope of type systems to address a wider range of safety properties. Representative examples include ESC/Java (Flanagan et al. 2002), Epigram (McBride and McKinna 2004), Spec# (Barnett et al. 2004), ATS (Chen and Xi 2005), Concoqtion (Pasalic et al. 2007), Sage (Gronski et al. 2006), Agda (Norell 2007), and Ynot (Nanevski et al. 2008). Each of these systems integrates some form of specification logic into the type system in order to rule out a wider range of truly bad states.

However, in the case of ESC/Java, Spec#, and Sage, the program logic is too weak to support full verification because these systems rely completely upon provers to discharge verification conditions automatically. While there have been great advances in the performance of automated provers, in practice, they can only handle relatively shallow fragments of first-order logic. Thus, programmers are frustrated when correct code is rejected by the type-checker. For example, none of these systems is able to prove that an array index is in bounds when the constraints step outside quantifier-free linear arithmetic.

In contrast, Agda, ATS, Concoqtion, Epigram, and Ynot use powerful, higher-order logics that support a much wider range of policies including (partial) correctness. Furthermore, in the case of Ynot, programmers can define and use connectives in the style of separation logic (Reynolds 2002) to achieve simple, modular specifications of higher-order imperative programs. For example, a recent paper (Nanevski et al. 2008) coauthored by some of the present authors describes how Ynot was used to construct fully-verified implementations of data structures such as queues, hash tables, and splay trees, including support for higher-order iterators that take effectful functions as arguments.

The price paid for these more powerful type systems is that, in general, programmers must provide explicit proofs to convince the type-checker that code is correct. Unfortunately, explicit proofs can be quite large when compared to the code. For example, in the Ynot code implementing dequeue for imperative queues, only 7 lines of program code are required, whereas the proof of correctness is about 70 lines.

This paper reports our experience re-designing and re-implementing Ynot to dramatically reduce the burden of writing and maintaining the necessary proofs for full verification. Like the original Ynot, our system is based on the ideas of Hoare Type Theory (Nanevski et al. 2006) and is realized as an axiomatic extension of the Coq proof assistant (Bertot and Castéran 2004). This allows us to inherit the full power of Coq's dependent types for writing code, specifications, and proofs, and it allows us to use Coq's facility for extraction to executable ML code. However, unlike in the previous version, we have taken advantage of Coq's tactic language, Ltac, to implement a set of parameterized procedures for automatically discharging, or at least simplifying, the separation logic-style verification conditions. The careful design of these procedures makes it possible for programmers to teach the prover about new domains as they arise.

We describe this new implementation of Ynot and report on our experience implementing and verifying various imperative data structures including stacks, queues, hash tables, binomial trees, and binary search trees. When compared with the previous version of Ynot, we observe roughly an order of magnitude reduction in proof size. In most cases, to realize automation, programmers need only prove key lemmas regarding the abstractions used in their interfaces and plug these lemmas into our extensible tactics. Additionally, we show that the tactics used to generate the proofs are robust to small changes in the code or specifications.

In the next section, we introduce the new Ynot in tutorial style. Next, we describe the automation tactics that we built, report on further evaluation of our system via case studies, compare with related work, and conclude.

1.1 Coq as an Extensible Automated Theorem Prover

Almost everyone familiar with Coq associates it with a particular style of proof development, which might be called the "video game" approach, after a comment by Xavier Leroy. A theorem is proved in many steps of manual interaction, where Coq tells the user which goals remain to be proved, the user enters a short command that simplifies the current goal somewhat, and the process repeats until no goals remain. One of our ancillary aims in this paper is to expose a broad audience to a more effective proof style. Coq provides very good support for fully automatic proving, via its domain-specific programming language Ltac (Delahaye 2000). This support can be mixed-and-matched with more manual proving, and it is usually the case that a well-written development starts out more manual and gradually transforms to a final form where no sequential proof steps are spelled out beyond which induction principle to use. Proof scripts of that kind often adapt without change to alterations in specifications and implementations.

We believe that awareness of this style is one of the crucial missing pieces blocking widespread use of proof assistants. We hope that the reader will agree that some of the examples that follow provide evidence that, for programmers with a few years of training using proof assistants, imperative programming with correctness verification need not be much harder than programming in Haskell.

2. The Ynot Programming Environment

To a first approximation, Coq can be thought of as a functional programing language like Haskell or ML, but with support for dependent types. For instance, one can have operations with types such as:

```
div : nat -> forall n : nat, n <> 0 -> nat
```

which uses dependency to capture the fact that `div` can only be called when a proof can be supplied that the second argument is non-zero. One can also write functions such as:

```
Definition avg (x:list nat) : nat :=
  let sum := fold plus 0 x in
  let len := length x in
  match eq_nat_dec len 0 with
    | inl(pf1: len = 0) => 0
    | inr(pf2: len <> 0) => div sum len pf2
  end.
```

This function averages the values in a list of natural numbers. It has a normal type like you might find in ML, and its implementation begins in an ML-like way, using a higher-order `fold` function. The interesting part is the `match` expression. We match on the result of a call to `eq_nat_dec`, a dependently-typed natural number comparison function. This function returns a sum type with an equality proof in one branch and an inequality proof in the other. We bind a name for each proof explicitly in the pattern for each `match` case. The proof that `len` is not zero is passed to `div` to justify the safety of the operation.

All Coq functions have to be pure – terminating without side effects. This is necessary to ensure that proofs really are proofs, with no spurious invalid "proofs by infinite loop." Ynot extends Coq with support for side-effecting computations. Similarly to Haskell, we introduce a monadic type constructor `ST T` which describes computations that might diverge and that might have side effects, but that, if they do return, return values of type `T`. The `ST` type family provides a safe way to keep the effectful computations separate from the pure computations.

Unlike Haskell's `IO` monad, the `ST` type family is parameterized by a pre- and post-condition, which can be used to describe the effects of the computation on a mutable store. Alternatively, one can think of the axiomatic base of Ynot as a fairly standard Hoare logic. The main difference of our logic from usual presentations is that it is designed to integrate well with Coq's functional programming language. Therefore, instead of defining a language of commands, we formalize a language of expressions in the style of the IO monad. A program derivation is of the form $\{P\}\ e\ \{Q\}$, where P is a pre-condition predicate over heaps, and Q is a post-condition predicate over an initial heap, the value that results from evaluating e, and a final heap. For instance, where we write `sel` and `upd` for the heap selection and update operations used in the ESC tools (Flanagan et al. 2002), we can derive the following facts, where p_1 and p_2 are pointer variables bound outside of the commands that we are verifying.

$$\{\lambda\_.\ \top\}\ \mathsf{return}(1)\ \{\lambda h, v, h'.\ h' = h \wedge v = 1\}$$

and

$$\begin{gathered}\{\lambda h.\mathsf{sel}(h, p_1) = p_2\}\\ x \leftarrow !p_1; x := 1\\ \{\lambda h, \_, h'.\ \mathsf{sel}(h, p_1) = p_2 \wedge h' = \mathsf{upd}(h, p_2, 1)\}\end{gathered}$$

Unlike other systems, Ynot does not distinguish between programs and derivations. Rather, the two are combined into one dependent type family, whose indices give the specifications of programs. For instance, the type of the "return" example would be:

```
ST (fun _ => True) (fun h v h' => h' = h /\ v = 1)
```

Heaps are represented as functions from pointers to dynamically-typed packages, which are easy to implement in Coq with an inductive type definition. The pointer read rule enforces that the heap value being read has the type that the code expects. The original Ynot paper (Nanevski et al. 2008) contains further details of the base program logic.

$$\frac{}{\{\mathbf{emp}\}\ \mathsf{return}(v)\ \{\lambda v'.\ [v = v']\}} \qquad \frac{\{P_1\}\ e_1\ \{Q_1\} \quad (\forall x, \{P_2(x)\}\ e_2\ \{Q_2\}) \quad (\forall x, Q_1(x) \Rightarrow P_2(x))}{\{P_1\}\ x \leftarrow e_1; e_2\ \{Q_2\}}$$

$$\frac{}{\{\mathbf{emp}\}\ \mathsf{new}(v)\ \{\lambda p.\ p \mapsto v\}} \qquad \frac{}{\{\exists v, p \mapsto v\}\ \mathsf{free}(p)\ \{\lambda\_.\ \mathbf{emp}\}}$$

$$\frac{}{\{\exists v, p \mapsto v * P(v)\}\ !p\ \{\lambda v.\ p \mapsto v * P(v)\}} \qquad \frac{}{\{\exists v, p \mapsto v\}\ p := v'\ \{\lambda\_.\ p \mapsto v'\}}$$

$$\frac{P \Rightarrow P' \quad \{P'\}\ e\ \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\ e\ \{Q\}} \qquad \frac{\{P\}\ e\ \{Q\}}{\{P * R\}\ e\ \{Q * R\}}$$

Figure 1. The main rules of the derived separation logic

2.1 A Derived Separation Logic

Direct reasoning about heaps leads to very cumbersome proof obligations, with many sub-proofs that pairs of pointers are not equal. Separation logic (Reynolds 2002) is the standard tool for reducing that complexity. The previous Ynot system built a separation logic on top of the axiomatic foundation, and we do the same here. We introduce no new inductive type of separation logic formulas. Instead, we define functions that operate on arbitrary predicates over heaps, with the intention that we will only apply these functions on separation-style formulas. Nonetheless, it can be helpful to think of our assertion language as defined by:

$$P \quad ::= \quad [\phi] \mid x \mapsto y \mid P * P \mid \exists x, P$$

For any pure Coq proposition ϕ, $[\phi]$ is the heap predicate that asserts that ϕ is true and the heap is empty. We write $\mathbf{emp}$ as an abbreviation for $[\mathsf{True}]$, which asserts only that the heap is empty. $x \mapsto y$ asserts that the heap contains only a mapping from x to y. $P_1 * P_2$ asserts that the heap can be broken into two heaps h_1 and h_2 with disjoint domains, such that h_1 satisfies P_1 and h_2 satisfies P_2. The final clause provides existential quantification.

The embedding in Coq provides much more expressive formulas than in most systems based on separation logic. Not only can any pure proposition be injected with $[\cdot]$, but we can also use arbitrary Coq computation to build impure assertions. For instance, we can model deterministic disjunction with pattern-matching on values of algebraic datatypes, and we can include calls to custom recursive functions that return assertions. We need no special support in the assertion language to accommodate this, and Coq's theorem-proving support for reasoning about pattern-matching recursive functions can be used without modification.

If we had defined an inductive type of specifications, we would have needed to encode most of the relevant Coq features explicitly. For instance, to allow pattern matching that produces specifications, our inductive type would need a constructor standing for dependent pattern matching, which is quite a tall order on its own.

Perhaps surprisingly, we have met with general success in implementing realistic examples using just these connectives. Standard uses of other connectives can often be replaced by uses of higher-order features, and the connectives that we *do* use are particularly amenable to automation. In Section 2.2, we try to give a flavor of how to encode disjunction, in the context of a particular example. Fully-automated systems like Smallfoot (Berdine et al. 2005) build in restrictions similar to ours, but it surprised us that we needed little more to do full correctness verification.

2.1.1 The Importance of Computational Irrelevance

What we have described so far is the same as in the original Ynot work. The primary departure of our new system is that we use a more standard separation logic. The old Ynot separation logic used *binary post-conditions* that may refer to both the initial and final heaps. (In both systems, specifications may refer to computation result values, so we avoid counting those in distinguishing between "unary" and "binary" post-conditions.) This is in stark contrast to traditional separation logics, where all assertions are separation formulas over a single heap, and all verification proof obligations are implications between such assertions. The utility of this formalism has been born out in the wealth of tools that have used separation logic for automated verification. In contrast, proofs of the binary post-conditions in the old Ynot tended to involve at least tens of steps of manual proof per line of program code. Today, even pencil-and-paper proofs about relationships between multiple heaps can draw on no logical formalism that comes close to separation logic in crispness or extent of empirical validation. While binary post-conditions are strictly more expressive than unary post-conditions, the separation logic community has developed standard techniques for mitigating the problem.

To make up for this lost expressiveness, we need, in effect, to move to a richer base logic. The key addition that lets us use a more standard formulation is the feature of *computationally-irrelevant variables*, which correspond to specification variables (also known as "ghost variables") in standard separation logic. Such variables may be mentioned in assertions and proofs only, and an implementation must enforce that they are not used in actual computation. Coq*, a system based on the Implicit Calculus of Constructions (Barras and Bernardo 2008), supports this feature natively. From a theoretical standpoint, it would be cleanest to implement Ynot as a Coq* library. However, in implementing the original Ynot system, we hesitated to switch to this nonstandard branch of the Coq development tree. In designing the new system, we felt the same trepidation, since we might encounter difficulties using libraries written for the standard Coq system, and the users of our library would need to install an unusual version of Coq. We hope that, in the long term, the new Coq* features will become part of the standard Coq distribution.

For now, we use an encoding of computationally-irrelevant variables that is effective in standard Coq, modulo some caveats that we discuss below. Our reimplementation employs the trick of representing specification variables in types that are marked as "proofs" instead of "programs," such that we can take advantage of Coq's standard restrictions on "information flow" from proofs to programs. Concretely, the Coq standard library has for some time contained a type family called `inhabited`, defined by:

```
Inductive inhabited (A:Type) : Prop :=
  inhabits : A -> inhabited A.
```

This code demonstrates Coq's standard syntax for inductive type definitions, which is quite similar to the syntax for algebraic datatype definitions in ML and Haskell. This type family has one parameter `A` of type `Type`, which can be thought of as the type of all types[1]. The constructor `inhabits` lets us inject any value into `inhabited`. While the original value may have an arbitrary type, the `inhabited` package has a type in the universe `Prop`, the universe of logical propositions. Terms whose types are in this universe are considered to be proofs and are erased by program extraction.

We will see in the following examples how this encoding necessitates some mildly cumbersome notation around uses of irrelevant variables. Further, to reason effectively about irrelevant variables, we need to assert without proof an axiom stating that the constructor `inhabits` is injective.

```
Axiom pack_injective : forall (T : Set) (x y : T),
  inhabits x = inhabits y -> x = y.
```

Our library additionally assumes the standard axiom of function extensionality ("functions are equal if they agree at all inputs") and the very technical "unicity of equality proofs" axiom that is included in Coq's standard library. This pair of axioms has been proved consistent for Coq's logic, and we could avoid appealing to extensionality at the cost of more proving work in the library, by formalizing heaps as lists instead of functions. Such a change would be invisible to most users of the library, who only need to use standard theorems proved about the heap model.

However, the pack injectivity axiom contradicts the axiom of proof irrelevance (which we do not use in any of our developments, but which is popular among Coq users), and it is an open question in the Coq community whether this axiom is consistent with Coq's logic even by itself. Past work built a denotational model for Ynot minus this feature (Petersen et al. 2008), and the architects of that model are now considering how to add irrelevance, which would complete the foundational story for the framework that we use in this paper. We hope that the experiences we report here can help to justify the inclusion of irrelevance as a core Coq feature.

2.1.2 The Rules of the Separation Logic

Figure 1 presents the main rules of our separation logic. The notable divergence from common formulations is in the use of existential quantifiers in the rules for freeing, reading, and writing. These differences make sense because Ynot is implemented within a constructive logic. Coq's constructivity is inspired by the Curry-Howard isomorphism, where programs and proofs can be encoded in the same syntactic class. A more standard, classical separation logic would probably require that, in the rule for free, the value v pointed to by p be provided as an argument to the proof rule. In constructive logic, such a value can only be produced when it can be computed by an algorithm, just as a functional program may only refer to a value that it has said how to compute. Additionally, we would not be able to use any facts implied by the current heap assertion to build one of these rule witnesses, and perhaps the witness can only be proved to exist using such facts. The explicit existential quantifier frees us to reason inside the assertion language in finding the witness.

Because it uses quantification in this way, the "read" rule must also take a kind of explicit framing condition. This condition is parameterized by the value being read from the heap, making it a kind of description of the neighborhood around that value in the heap. More standard separation logics force the exact value being read to be presented as an argument to the proof rule, but here we want to allow verification of programs where the exact value to read cannot be computed from the pieces of pure data that are in scope.

We want to emphasize that the changes we have made in the Ynot separation logic have no effect on the theory behind the systems. In both the old and new systems, a separation logic is defined on top of the base Hoare logic with binary post-conditions, introducing no new axioms. Here, we use the same base logic as in the past work, so the past investigations into its metatheory (Petersen et al. 2008) continue to apply. The sole metatheoretical wrinkle is the one which we discussed above, involving computational irrelevance, which is orthogonal to program logic rules.

In the rest of this section, we will introduce the Ynot programming environment more concretely, via several examples of verified data structure implementations.

```
Module Type STACK.
  Parameter t : Set -> Set.
  Parameter rep (T : Set) : t T -> list T -> hprop.

  Parameter new T :
    Cmd emp (fun s : t T => rep s nil).
  Parameter free T (s : t T) :
    Cmd (rep s nil) (fun _ : unit => emp).

  Parameter push T (s : t T) (x : T)
    (ls : [list T]) :
    Cmd (ls ~~ rep s ls)
        (fun _ : unit => ls ~~ rep s (x :: ls)).
  Parameter pop T (s : t T) (ls : [list T]) :
    Cmd (ls ~~ rep s ls)
        (fun xo : option T => ls ~~
          match xo with
            | None => [ls = nil] * rep s ls
            | Some x => Exists ls' :@ list T,
                          [ls = x :: ls']
                          * rep s ls'
          end).
End STACK.
```

Figure 2. The signature of an imperative stack module

2.2 Verifying an Implementation of Imperative Stacks

Figure 2 shows the signature of a Ynot implementation of the stack ADT. The signature is expressed in Coq's ML-like module system. Each implementation contains a type family `t`, where, for any type `T`, a value of `t(T)` represents a stack storing elements of `T`. The `rep` component of the interface relates an imperative stack `s` to a functional list `ls` in a particular state. Thus, `rep s ls` is a predicate on heaps (`hprop`) which can be read as "`s` represents the list `ls`" in the current state. Just as abstraction over the type family `t` allows an implementation to choose different data structures to encode the stack, abstraction over the assertion `rep` allows an implementation to choose different invariants connecting the concrete representation to an idealized model.

In Section 2.1, we gave a grammar for our "specification language." In contrast to most work on separation logic, our real implementation has no such specification language. Rather, we define the type `hprop` as `heap -> Prop`, so that specifications and invariants are arbitrary predicates over heaps. In Figure 2, we see notations involving `emp`, asserting that the heap is empty; `[...]`, for injecting pure propositions; `*`, for the standard separating conjunction; and `Exists`, for standard typed existential quantification.

[1] To avoid the standard soundness problems with including a type of all types, actual Coq type-checking infers numerical indices for all occurrences of `Type`.

Not shown in this figure is the binary "points-to" operator `-->`. The relative parsing precedences of the operators place `-->` highest, followed by `*` and `Exists`. We only need to use funny symbols for syntax like `Exists x :@ T, P` (meaning "there exists `x` of type `T` such that `P`") to avoid confusing the LL(1) parser that is at the heart of Coq's syntax extension facilities. Our library defines `hprop`-valued functions implementing these usual separation logic connectives, but users can define their own "connectives" just as easily. For example, here is how we define `Exists`:

```
Definition hprop_ex (T : Type) (p : T -> hprop) :=
  fun h : heap => exists v : T, p v h.
```

Here is how we add a syntax extension (or "macro") that lets us write existential quantification in the way seen in Figure 2:

```
Notation "'Exists' v :@ T , p" :=
  (hprop_ex T (fun v : T => p)).
```

By reading the types of the methods exposed in the `STACK` signature, we can determine the contract that each method adheres to. The `Cmd` type family is our parameterized monad of computations with separation logic specifications; the two arguments to `Cmd` give preconditions and postconditions. `Cmd` is defined in terms of the more primitive `ST` parameterized monad, in the same way as in our past work (Nanevski et al. 2008)[2]. Our specifications follow the algebraic approach to proofs about data abstraction (as in Liskov and Zilles (1975)), where an abstract notion of state is related to concrete states. Each operation needs a proof that it preserves the relation properly. In Ynot developments, abstract states are manipulated by standard, purely-functional Coq programs, and method specifications include explicit calls to these state transformation functions. Each post-condition requires that the new concrete, imperative state be related to the abstract state obtained by transforming the initial abstract state.

The type of the `new` operation tells us that it expects an empty heap on input, and on output the heap contains just whatever mappings are needed to satisfy the representation invariant between the function return value and the empty list. The `free` operation takes a stack `s` as an argument, and it expects the heap to satisfy `rep` on `s` and the empty list. The post state shows that all heap values associated with `s` are freed.

The specification for `push` says that it expects any valid stack as input and modifies the heap so that the same stack that stood for some list `l` beforehand now stands for the list `x :: l`, where `x` is the appropriate function argument. We see an argument `ls` with type `[list T]`. The brackets are a notation defined by the Ynot library, standing for computational irrelevance. The syntax `[T]` expands to `inhabited T`. To review our discussion from Section 2.1.1, this means that the type-checker should enforce that the value of `ls` is not needed to execute the function. Rather, such values may only be used in stating specifications and discharging proof obligations. We use Coq's notation scope mechanism to overload brackets for writing irrelevant types and lifted pure propositions.

For an assertion `P` that mentions the irrelevant variable `v`, the notation `v ~~ P` must be used to unpack `v` explicitly. The type of the unpack operation is such that it may only be applied to assertions and may not be used to allow an irrelevant variable's value to leak into the computational part of a program. Unpacking has no "logical" meaning; it is only used to satisfy the type-checker in the absence of native support for irrelevance. The notation is defined by this equation, where we write "`[v'/v]`" informally to denote the substitution of variable `v'` for variable `v` in a Coq term.

```
v ~~ P = (exists v', v = inhabits v' /\ P[v'/v])
```

[2] The derived monad is called "STsep" in that past work.

The type of `pop` showcases how we avoid the disjunctive connectives of separation logic. The function returns an optional `T` value, which will be `None` when the stack is empty and will be `Some x` when `x` is at the top of the stack. We use a Coq `match` expression to give a different post-condition for each case.

We can implement a module satisfying this signature. With the type `T` as a local variable, we can define the type of nodes of the linked lists that we will use. We use the abstract type `ptr` of untyped pointers from the Ynot library.

```
Record node : Set := Node {
  data : T;
  next : option ptr
}.
```

To define the representation invariant, we want a recursive function specifying what it means for a possibly-null pointer to represent a functional list. Our code contains a `struct` annotation that gives a termination argument for the function.

```
Fixpoint listRep (ls : list T) (hd : option ptr)
    {struct ls} : hprop :=
  match ls with
    | nil => [hd = None]
    | h :: t => match hd with
                  | None => [False]
                  | Some hd => Exists p :@ option ptr,
                     hd --> Node h p * listRep t p
                end
  end.
```

We can represent stacks as untyped pointers to the heads of linked lists built from `Node`s.

```
Definition stack := ptr.
```

We achieve type safety through the representation invariant.

```
Definition rep (s : stack) (ls : list T) : hprop :=
  Exists po :@ option ptr, s --> po * listRep ls po.
```

Before we start implementing the ADT methods, we should set up some proof automation machinery. Systems like Smallfoot (Berdine et al. 2005) have hardcoded support for particular heap predicates like acyclic linked list-ness, cyclic linked list-ness, and so on. These systems perform simplifications on formulas that mention the predicates that they understand. In Ynot, on the other hand, the programmer can define his own new predicates, as we have just done. Not only that, but he can also prove lemmas that correspond to the simplification rules built into automated tools, and he can plug his lemmas into a general separation logic solver. All of this is done with no risk that a mistake by the programmer will lead to a faulty verification; every lemma must be proved from first principles.

In a real proof, of course, the human proof architect only learns which automation will be effective in the course of verifying his program. Ynot supports this kind of incremental automation very well, as we hope to demonstrate in the rest of the section, using sample interactive Coq sessions. Due to space constraints, we must skip some steps and go straight to the right answers, but we have tried to include enough iteration to give a flavor for Ynot development.

As we progress through the methods, we will be improving a custom tactic, or proof procedure, that we will design specifically for this data structure, and we will call that tactic `tac`. We begin with a version of `tac` that delegates all work to the separation logic simplifier `sep` that is included with Ynot.

```
Ltac tac := sep fail auto.
```

We will explain each of the two parameters to sep as we find a use for it. We implement each stack method by stating its type as a proof search goal, using tactics to realize the goal step by step. The first method to implement is new, and we do so using the syntax New for the new(·) operation from Figure 1.

```
Definition new : Cmd emp
    (fun s : stack => rep s nil).
  refine {{New None}}; tac.
```

A simple two-step proof script should suffice. We first use the refine tactic to provide a template for the implementation. The template may have holes in it, and each hole is added as a subgoal. We chain our tac tactic with the semicolon operator, so that tac is applied to each subgoal generated from a hole.

Here, we see no proof holes to be filled in, but some are nonetheless there, hidden by the notation {{...}}, which we define in a Coq source file in our library, using Coq's syntax extension mechanism:

```
Notation "{{ st }}" :=
  (SepWeaken _ (SepStrengthen _ st _) _).
```

This rule requests that every use of the double braces be expanded using the template on the second line, leaving four holes to be filled. The SepWeaken and SepStrengthen functions are for weakening post-conditions and strengthening pre-conditions, and the four holes, in order, are to be filled by a new post-condition, a new pre-condition, a proof that the new pre-condition implies the old, and a proof that the old post-condition implies the new. In the case of the new method, the new specifications are determined by standard type inference, while the two proofs must be added as new goals. With the proof script we have used so far, one proof goal remains in the definition of new and is shown to the user:

```
v --> None ==> rep v nil
```

The syntax ==> is for implication between heap assertions, and it has lower parsing precedence than any of the other operators that we use. We see that it is important to unfold the definition of the representation predicate, so we modify our tactic definition, and now the proof completes automatically.

```
Ltac tac := unfold rep; sep fail auto.
```

The definitions of free and push are not much more complicated. We use some new notations, including a Haskell-inspired monadic bind syntax, and all are defined in our library with "Coq macros," as in the example of double braces above.

```
Definition free (s : stack) : Cmd (rep s nil)
    (fun _ : unit => emp).
  refine (fun s => {{Free s}}); tac.
Qed.
```

```
Definition push (s : stack) (x : T) (ls : [list T])
    : Cmd (ls ~~ rep s ls)
      (fun _ : unit => ls ~~ rep s (x :: ls)).
  refine (fun s x ls => hd <- !s;
    nd <- New (Node x hd);
    {{s ::= Some nd}}
  ); tac.
Qed.
```

The implementation of pop uses another syntax extension, which provides an IfNull expression form. The option-typed argument to IfNull is checked for nullness (i.e., equality to None). In an Else branch, where the pointer is known to be non-null, that fact is added as a usable proof hypothesis, and the variable being tested is rebound with a non-option type. We use ;; instead of ; after imperative commands that do not bind variables, because attempts to do otherwise confuse Coq's finicky LL(1) parser.

```
Definition pop (s : stack) (ls : [list T]) :
  Cmd (ls ~~ rep s ls) (fun xo : option t => ls ~~
        match xo with
          | None => [ls = nil] * rep s ls
          | Some x => Exists ls' :@ list T,
                        [ls = x :: ls'] * rep s ls'
        end).
  refine (fun s ls => hd <- !s;
    IfNull hd Then
      {{Return None}}
    Else
      nd <- !hd;
      Free hd;;
      s ::= next nd;;
      {{Return (Some (data nd))}}); tac.
```

Several unproved subgoals are returned, this one among them, containing a unification variable ?1960:

```
s --> Some hd0 * listRep x (Some hd0)
  ==> hd0 --> ?1960 * hd0 --> ?1960
```

We can tell that something has probably gone wrong, since the conclusion of the implication contains an unsatisfiable separation formula that mentions the same pointer twice. Our automated separation simplification is quite aggressive and often simplifies satisfiable formulas to unsatisfiable forms, but the results of this process tend to provide hints about which facts would have been useful. In this case, we see a use of listRep where the pointer is known to be non-null. We can prove a lemma that would help simplify such formulas.

```
Theorem listRep_Some :
  forall (ls : list T) (hd : ptr),
  listRep ls (Some hd) ==> Exists h :@ T,
    Exists t :@ list T, Exists p :@ option ptr,
      [ls = h :: t] * hd --> Node h p * listRep t p.
  destruct ls; sep fail ltac:(try discriminate).
Qed.
```

We prove that a functional list related to a non-null pointer decomposes in the expected way. All it takes is for us to request a case analysis on the variable ls, followed by a call to the separation solver. Here we put to use the second parameter to sep, which gives a tactic to try applying throughout proof search. The discriminate tactic solves goals whose premises include inconsistent equalities over values of datatypes, like nil = x :: ls; and adding try in front prevents discriminate from signaling an error if no such equality exists.

We can modify our tac tactic to take listRep_Some into account. First, we define another procedure for simplifying an implication.

```
Ltac simp_prem := simpl_IfNull;
  simpl_prem ltac:(apply listRep_Some).
```

Our tactic first calls a simplification procedure associated with the IfNull syntax extension. Next, our procedure calls a tactic simpl_prem from the Ynot library, for simplifying premises of implications. The argument to simpl_prem gives a procedure to attempt on each premise, until no further progress can be made.

We can redefine tac to use simp_prem, by passing that new procedure as the first argument to sep. That first argument is used by sep to simplify a goal before beginning the main proof search.

```
let pop s =
  sepBind (sepStrengthen (sepRead s)) (fun hd ->
    match hd with
      | Some v ->
          sepBind (sepStrengthen (sepRead v))
            (fun nd ->
            sepSeq (sepStrengthen (sepFrame
                (sepFree v)))
              (sepSeq (sepStrengthen (sepFrame
                  (sepWrite s (next nd))))
                (sepWeaken
                  (sepStrengthen (sepFrame
                    (sepReturn (Some
                      (data nd))))))))
      | None -> sepWeaken (sepStrengthen (sepFrame
                  (sepReturn None))))
```

Figure 3. Sample OCaml code extracted from the stack example

We also suggest to sep that try discriminate may be useful throughout proof search.

```
Ltac tac := unfold rep; sep simp_prem
                              ltac:(try discriminate).
```

When we rerun the definition of pop, we have made progress. Only one goal remains to prove:

```
emp ==> [x = nil]
```

We see that this goal probably has to do with a case where we know that the list being modeled is nil. We were successful at using simpl_prem to deal with the case where we know the list is non-nil, and we can continue with that strategy by proving another lemma.

```
Theorem listRep_None : forall ls : list T,
  listRep ls None ==> [ls = nil].
  destruct ls; sep fail idtac.
Qed.
```

Now our verification of pop completes, after we modify the definition of simp_prem:

```
Ltac simp_prem := simpl_IfNull;
  simpl_prem ltac:(apply listRep_None
                 || apply listRep_Some).
```

We complete the implementation of the stack ADT with a trivial definition of the type family t, relying on the representation invariant to ensure proper use.

```
Definition t (_ : Set) := stack.
```

For our modest efforts, we can now extract an executable OCaml version of our module. Figure 3 shows part of the result of running Coq's automatic extraction command on our Stack module.

In the implementation of pop, we see invocations of functions whose names begin with sep. These come from the Ynot library, and we must provide their OCaml implementations. Any Ynot program that returns a type T may be represented in unit -> T in OCaml, regardless of the specification appearing in the original Coq type. This makes it easy to implement the basic functions, in the spirit of how the Haskell IO monad is implemented. We see calls to explicit weakening, strengthening, and framing rules in the extracted code. In OCaml, these can be implemented as no-ops and erased by an optimizer.

Notice that all specification variables and proofs are eliminated automatically by the Coq extractor. With the erasure of weakening and related operations, we arrive at exactly the kind of monadic code that is standard fare for Haskell, such that the compilation techniques developed for Haskell can be put to immediate use in creating an efficient compilation pipeline for Ynot.

It is also worth pointing out that the sort of tactic construction effort demonstrated here is generally per data structure, not per program. We can verify a wide variety of other list-manipulating programs using the same tac tactic that we developed here. Usually, the tactic work for a new data structure centers on identifying the kind of unfolding lemmas that we proved above.

2.3 Verifying Imperative Queues

It is not much harder to implement and verify a queue structure. We define an alternate list representation, parameterized by head and tail pointers.

```
Fixpoint listRep (ls : list T) (hd tl : ptr)
    {struct ls} : hprop :=
  match ls with
    | nil => [hd = tl]
    | h :: t => Exists p :@ ptr,
                  hd --> Node h (Some p)
                  * listRep t p tl
  end.

Record queue : Set := Queue {
  front : ptr;
  back : ptr
}.

Definition rep' (ls : list T) (fr ba : option ptr) :=
  match fr, ba with
    | None, None => [ls = nil]
    | Some fr, Some ba => Exists ls' :@ list T,
        Exists x :@ T, [ls = ls' ++ x :: nil]
          * listRep ls' fr ba * ba --> Node x None
    | _, _ => [False]
  end.

Definition rep (q : queue) (ls : list T) :=
  Exists fr :@ option ptr, Exists ba :@ option ptr,
    front q --> fr * back q --> ba * rep' ls fr ba.
```

For this representation, we prove similar unfolding lemmas to those we proved for stacks, with comparable effort. We also need a new lemma for unfolding a queue from the back.

```
Lemma rep'_back : forall (ls : list T) (fr ba : ptr),
  rep' ls (Some fr) ba
  ==> Exists nd :@ node, fr --> nd
    * Exists ls' :@ list T, [ls = data nd :: ls']
      * match next nd with
          | None => [ls' = nil]
          | Some fr' => rep' ls' (Some fr') ba
        end.
```

The proof of the lemma relies on some lemmas about pure functional lists. With those available, we prove rep'_back in under 20 lines. When we plug this and the two other unfolding lemmas into the sep procedure, we arrive at quite a robust proof procedure for separation assertions about lists that may be modified at either end.

Again, in our final queue implementation, every proof obligation is proved by a tac tactic built from sep. We write under 10 lines of new tactic hints to be applied during proof search, and we must prove one key lemma by induction. We discover the importance of this lemma while trying to verify an implementation of enqueueing.

```
Definition enqueue :
    forall (q : queue) (x : T) (ls : [list T]),
    Cmd (ls ~~ rep q ls)
    (fun _ : unit => ls ~~ rep q (ls ++ x :: nil)).
  refine (fun q x ls => ba <- !back q;
    nd <- New (Node x None);
    back q ::= Some nd;;
    IfNull ba Then
      {{front q ::= Some nd}}
    Else
      ban <- !ba;
      ba ::= Node (data ban) (Some nd);;
      {{Return tt}}); tac.
```

Coq returns a single unproved goal:

```
listRep v2 v4 p * p --> Node v3 (Some nd)
  ==> listRep (v2 ++ v3 :: nil) v4 nd
```

Considering this goal, we see that the problem is that it can only be proved by induction. In general, we must be explicit about induction everywhere we need it, so we need to prove a lemma about this case. The lemma itself is quite easy to automate, when we add one hint from the Ynot library about the commutativity of separating conjunction.

```
Lemma push_listRep : forall (ba : ptr) (x : T)
  (nd : ptr) (ls : list T) (fr : ptr),
  ba --> Node x (Some nd) * listRep ls fr ba
  ==> listRep (ls ++ x :: nil) fr nd.
  Hint Resolve himp_comm_prem.
  induction ls; tac.
Qed.
```

To get the original verification to go through, we only need to add this lemma to the hint database, using a built-in Coq command.

```
Hint Immediate push_listRep.
```

2.4 Loops

Like with many semi-automated verification systems, we require annotations that are equivalent to loop invariants. Since Coq's programming language is functional, it is more natural to write loops as recursive functions, and the loop invariants become the pre- and post-conditions of these functions.

We support general recursion with a primitive fixpoint operator in the base program logic, and it is easy to build a separation logic version on top of that. We can also build multiple-argument recursive and mutually-recursive function forms on top of the single-argument form, without needing to introduce new primitive combinators.

An example is a `getElements` function, defined in terms of the list invariant that we wrote for the stack example. This operation returns the functional equivalent of an imperative list. The task is not so trivial as it may look at first, because the computational irrelevance of the function's second argument prohibits its use to influence the return value. This means that we are not allowed to name the irrelevant argument as one that decreases on each recursive call, which prevents us from using Coq's native recursive function definitions, where every function must be proved to terminate using simple syntactic criteria. Nonetheless, the definition is easy using the general recursion combinators supported by Ynot.

```
Definition getElements (hd : option ptr)
  (ls : [list A]) :
  Cmd (ls ~~ listRep ls hd)
      (fun res : list A => ls ~~ [res = ls]
        * listRep ls hd).
  refine (Fix2
    (fun hd ls => ls ~~ listRep ls hd)
    (fun hd ls res => ls ~~ [res = ls]
                      * listRep ls hd)
    (fun self hd ls =>
      IfNull hd Then
        {{Return nil}}
      Else
        fn <- !hd;
        rest <- self (next fn)
                     (ls ~~~ tail ls) <@> _;
        {{Return (data fn :: rest)}})); tac.
Qed.
```

```
Module Type MEMO.
  Parameter T : Set.
  Parameter t : forall (T' : T -> Set),
    hprop
    -> (forall x, T' x -> Prop)
    -> Set.
  Parameter rep : forall (T' : T -> Set)
    (inv : hprop) (fpost : forall x, T' x -> Prop),
    t inv fpost -> hprop.
  Parameter create : forall (T' : T -> Set)
    (inv : hprop) (fpost : forall x, T' x -> Prop),
    (forall x, Cmd inv
      (fun y : T' x => [fpost _ y] * inv))
    -> Cmd emp (fun m : t inv fpost => rep m).
  Parameter funcOf : forall (T' : T -> Set)
    (inv : hprop) (fpost : forall x, T' x -> Prop)
    (m : t inv fpost),
    forall (x : T), Cmd (rep m * inv)
      (fun y : T' x => rep m * [fpost _ y] * inv).
End MEMO.
```

Figure 4. Signature of a memoization module

The code demonstrates a use of one of the derived fixpoint combinators, `Fix2`. Of the three arguments that we pass, the first two give the pre-condition and post-condition in terms of the two "real" arguments (and, for the post-condition, the return value). The third argument is the function body. It takes a recursive self-reference as its first argument, followed by the two "real" arguments. Native Coq recursive function definitions must often include annotations explaining why they terminate, but Ynot deals only with partial correctness, so no such annotations are required for our fixpoint combinators.

The notation `x ~~~ e` is for building a new computationally-irrelevant value out of an old one. The notation `e <@> P` is explicit invocation of the frame rule. With the current system, one usually wants to invoke that rule at each function call. The framing assertion can be written as an underscore to ask that it be inferred.

2.5 A Dependently-Typed Memoizing Function

As far as we have been able to determine, all previous tools for data structure verification lack either aggressive automation or support for higher-order features. The original Ynot supported easy integration of higher-order functions and dependent types, but the very manual proof style became even more onerous for such uses. Our reimplemented Ynot maintains the original's higher-order features, and our proof automation integrates very naturally with them. This is a defining advantage of our new framework over all alternatives.

For instance, it is easy to define a module supporting memoization of imperative functions. Figure 4 gives the signature of our implementation, which is actually an ML-style functor that produces implementations of this signature when passed appropriate input modules. The type `T` is the domain of memoizable functions, and types like `t inv fpost` stand for memo tables. The argument `inv` is an assertion giving an invariant that the memoized function maintains, and the pure assertion `fpost` gives a relation between inputs and outputs of the function. The `rep` predicate captures the heap invariants associated with a memo table. The `create` function produces a memo table when passed an imperative function with the proper specification. Finally, the `funcOf` function maps a memo table to a function that consults the table to avoid recomputation.

We can implement a `MEMO` functor in 50 lines when we use a memo table that only caches the most recent input-output pair. Like in the previous examples, we build a specialized automation procedure with a one-line instantiation of library tactics. We give a 7-line definition of `rep`, give one one-liner proof of a lemma to use in proof search, and include two lines of annotations within the definition of `funcOf`. All of the rest of the development is no longer or more complicated than in ML. Compared to ML, we have the great benefit of using types to control the behavior of functions to be memoized. A function could easily thwart an ML memoizer by producing unexpected computational effects.

3. Tactic Support

The examples from the last two sections show how much of the gory details of proofs can be hidden from programmers. In actuality, every command triggers the addition of one or more proof obligations that cannot be discharged effectively by any of the built-in Coq automation tactics. Not only is it hard to prove the obligations, but it is also hard to infer the right intermediate specifications. Our separation logic formulas range well outside the propositional fragment that automated tools tend to handle; specification inference and proving must deal with higher-order features.

Here is an example of the proof obligations generated for the code we gave earlier for the stack `push` method. Numbers prefixed with question marks are unification variables, whose values the `sep` tactic must infer.

```
ls ~~ rep s ls ==>
Exists v :@ option ptr, s --> v * ?200 v

forall v : option ptr, s --> v * ?200 v ==> ?192 v

?192 hd ==> ?217 * emp

forall v : ptr, ?217 * v --> Node x hd ==> ?206 v

?206 nd ==> ?234 * (Exists v' :@ ?231, s --> v')

?234 * s --> Some nd ==> ls ~~ rep s (x :: ls)
```

We can see that each goal, compared to the previous goals, has at most one new unification variable standing for a specification; of the two new variables appearing in the second last line, one stands for a type, which will be easy to infer by standard unification, once the values of prior variables are known. Also, each new specification variable has its value determined by the value of the new variable from the previous goal. This is no accident; we designed our combinators and notations to have this property.

The effective range of specifications is too large to be solvable by any particular "magic bullet" tactic. Nonetheless, we have found that, in practice, a specific parameterized proof strategy can discharge most obligations. In contrast to the situation with classical verification tools that are backed by automated first-order theorem-provers, when any proof strategy fails in Coq, the user can always program his own new strategy or even move to mostly-manual proof. However, our experience suggests to us that most goals about data structures can be solved by the procedure that we present in this section.

That procedure is implemented as the `sep` tactic that we used in our examples. We do not have space to include the literal Coq code implementing it; we will outline the basic procedure instead. The implementation is in Coq's Ltac language (Delahaye 2000), a domain-specific, dynamically-typed language for writing proof-generating proof search procedures. All of the proof scripts we have seen so far are really Ltac programs. The full language includes recursive function definitions, which, along with pattern-matching on proof goals, makes it possible to code a wide variety of proof manipulation procedures.

As our examples have illustrated, `sep` takes two arguments, which we will call `unfolder` and `solver`. The task of `unfolder` is to simplify goals before specification inference, usually by unfolding definitions of recursive predicates, based on known facts about their arguments. The task of `solver` is to solve all of the goals that remain after generic separation logic reasoning is applied.

Coq comes with the standard tactic `tauto`, for proving propositional tautologies. There is a more general version of `tauto` called `intuition`, which will apply a user-supplied tactic to finish off sub-proofs, while taking responsibility for handling propositional structure on its own. The `intuition` tactic also exhibits the helpful behavior of leaving for the user any subgoals that it could not establish. `sep` is meant to be an analogue of `intuition` for separation logic. We also want it to handle easy instantiation of existential quantifiers, since they appear so often in our specifications.

We can divide the operation of `sep` into five main phases. We will sketch the workings of each phase separately.

3.1 Simple Constraint Solving

It is trivial to determine the proper value for any unification variable appearing alone on one side of the implication. For instance, given the goal

```
p --> x * q --> y ==> ?123
```

we simply set `?123` to `p --> x * q --> y`. Given the slightly more complicated goal

```
p --> x * q --> y ==> ?123 x
```

we abstract over `x` in the premise to produce `fun x' => p --> x' * q --> y`.

3.2 Intermediate Constraint Solving

When the trivial unification rules are not sufficient, we need to do more work. We introduce names for all existential quantifiers and computationally-irrelevant variables in the premise. For instance, starting with

```
m ~~ Exists v :@ T, p --> v * rep m v
==> ?123 * Exists x :@ T, p --> x
```

we introduce names to simplify the premise, leading to this goal:

```
p --> v' * rep m' v'
  ==> ?123 * Exists x :@ T, p --> x
```

Now we run the user's `unfolder` tactic, which might simplify some use of a definition. Let us assume that no such simplification occurs for this example. We notice that the points-to fact on the right mentions the same pointer as a fact on the left, so these two facts may be unified, implying `x = v'`. Canceling this known information, we are left with

```
rep m' v' ==> ?123
```

which is resolvable almost trivially. We cannot give `?123` a value that mentions the variables `m'` and `v'`, since we introduced them with elimination rules within our proof. These variables are not in scope at the point in the original program where the specification must be inserted. Instead, we remember how each local variable was introduced and re-quantify at the end, like this:

```
m ~~ Exists v :@ T, rep m v ==> ?123
```

Now the trivial unification is valid. The crucial part of this process was the matching of the two points-two facts. We have special-case rules for matching conclusion facts under quantifiers, for conclusions that match the pre-conditions of the read, write, and free rules. Beyond that, we apply cancellation of identical terms on the two sides of the implication, when those terms do not fall under the scopes of quantifiers. These simple rules seem to serve well in practice.

3.3 Premise Simplification

After specification inference, the next step is to simplify the premise of the implication. Any **emp** in the premise may be removed, and any lifted pure formula $[\phi]$ may be removed from the implication and added instead to the normal proof context. We also remove existential quantifiers and irrelevant variable unpackings in the same way as in the previous phase.

3.4 Conclusion Simplification

The main `sep` loop is focused on dealing with parts of the conclusion. We remove occurrences of **emp**, and we remove any pure formula $[\phi]$ that the user's `solver` tactic is able to prove. An existential formula `Exists x :@ T, P(x)` in the conclusion is replaced by `P(?456)`, for a fresh unification variable `?456`. When no more of these rules apply, we look for a pair of unifiable subformulas on the sides of the implication. All such pairs are unified and crossed out. This may determine the value of a variable introduced for an existential quantifier.

For instance, say we begin with this goal.

```
[m < 17] * p --> m
  ==> Exists x :@ nat, p --> x * [x < 42]
```

Premise simplification would move the initial impure fact into the normal proof context, leaving us with this.

```
p --> m ==> Exists x :@ nat, p --> x * [x < 42]
```

Conclusion simplification would introduce a name for the existentially-bound variable.

```
p --> m ==> p --> ?789 * [?789 < 42]
```

Next, conclusion simplification would match the two `p` points-to facts, since their pointers unify trivially.

```
emp ==> [m < 42]
```

This goal can be reduced to `emp ==> emp` by using the normal proof context to deduce the fact inside the brackets.

3.5 Standard Coq Automation

When `sep` has run out of rules to apply, the remaining subgoal is subjected to standard Coq automation. Propositional structure and calls to recursive functions are simplified where possible. `sep` ends by running a loop over those simplifications and the simplifications performed by the user's `solver` tactic, until no further progress can be made. Finally, `sep` discharges all goals of the form `P ==> P`, by reflexivity.

Every step of the overall process is implemented in Ltac, so that only a bug in Coq would allow `sep` to declare an untrue goal as true, no matter which customization the programmer provides. By construction, every step builds an explicit proof term, which can be validated afterward with an independent checker that is relatively simple, compared to the operation of all of the decision procedures that may have contributed to the proof.

4. Evaluation

We have used our environment to implement and verify several data structures, including the Stack and Queue examples that appeared in Section 2. We also follow the evaluation of our prior Ynot system in implementing a generic signature of imperative finite maps. We built three very different implementations: a trivial implementation based on pointers to heap-allocated functional association lists, an implementation based on binary search trees, and an implementation based on hash tables. Any of the implementations can be used interchangeably via ML-style functors, and their shared signature is phrased in terms of dependently-typed maps, where the type of data associated with a key is calculated from an arbitrary Coq function over that key. Our largest example, a packrat PEG parser (Ford 2004), uses these finite maps to cache intermediate results.

We also verified one more exotic data structure: binomial trees, which are tree structures with a non-trivial rule for determining how many pointers are stored at each node. This data structure is often applied in implementing priority queues. Our implementation is interesting in its use of a dependently-typed recursive function to characterize functional models of such trees.

Finally, we chose representative examples from two competing data structure verification systems, Smallfoot (Berdine et al. 2005) and Jahob (Zee et al. 2008), and reimplemented those examples in our new Ynot.

Figure 5 presents code size statistics for our case studies. "Program" code is code that is preserved by extraction. "Specs" are the pre- and post-conditions of every function defined in the module. The core of a Ynot module consists of heap representation "rep" code (e.g., the definitions named `rep` in our examples), along with proofs (e.g., `push_listRep`) and tactics (e.g., `simp_prem`) dealing with these representations. The annotations column counts the number of lines of programmer specified annotations (e.g., `<@>`). The total overhead column sums proofs, tactics, and annotations. We also present type-checking and proving times (in minutes and seconds), as measured on a 2.8 GHz Pentium D with 1 GB of memory. So far, we have not optimized our tactics for running time; they are executed by direct interpretation of programs in a dynamically-typed language.

Our previous version of Ynot placed a significant interactive proof burden on the programmer. The previous Ynot hash table, for instance, required around 320 explicit Coq tactic invocations. Each tactic invocation (indicated by a terminating "." in Coq) represents a manual intervention by the Ynot programmer. These invocations tended to be low-level steps, like choosing which branch of a disjunction to prove. As such, these proofs are brittle in the face of minor changes. In some previous Ynot developments, the ratio of manual proof to program text is over 10 to 1. For comparison, a large scale compiler certification effort (Leroy 2006) has reported a proof-to-code ratio of roughly 6 to 1.

In contrast, our new hash table requires only about 70 explicit tactic invocations. These invocations tend to be high level steps, like performing induction or invoking the `sep` tactic. We have

	Program	Specs	Rep	Proofs	Tactics	Annotations	Total Overhead	Time (m:s)
Stack	14	8	14	7	5	0	12	0:12
Queue	26	12	22	41	25	0	66	1:36
Ref to Functional Finite Map	8	16	2	2	2	0	4	0:05
Hash Table	34	21	6	70	38	34	142	0:45
BST Finite Map	31	16	6	22	8	4	34	1:35
Binomial Tree	19	12	13	0	9	7	16	2:33
Association List	48	34	17	41	51	10	102	3:10
Linked List Segments	84	34	19	91	208	7	306	2:15
Packrat PEG Parser	277	110	15	102	55	5	162	1:20

Figure 5. Breakdown of numbers of lines of different kinds of code in the case studies

observed that such tactic-based proofs are significantly easier to maintain.

We also made rough comparisons against two verification systems that do not support reasoning about first-class functions. The Jahob (Zee et al. 2008) system allows the specification and verification of recursive, linked data structures in a fragment of Java. We implemented an association list data structure that is included as an example in the Jahob distribution. Code-size-wise, the two implementations are quite similar. For instance, they both require around twenty lines of heap representation code, and they both require about a dozen lines of code for the lookup function's loop invariant. Our Ynot implementation uses explicit framing conditions in places where Jahob does not, but we speculate that we can probably remove these annotations with additional, custom automation.

Our second comparison is against the Smallfoot (Berdine et al. 2005) system, which does completely automated verification of memory safety via separation logic. We implemented Ynot versions of 10 linked list segment functions included with the Smallfoot distribution. In each case, the Ynot and Smallfoot versions differed by no more than a few lines of annotation burden.

5. Related Work

Considering the two automated systems that we just mentioned, Smallfoot uses a very limited propositional logic, and Jahob uses an undecidable higher-order logic. Many interesting program specifications cannot be written in Smallfoot's logic and cannot be proved to hold by Jahob's automated prover. Neither of these systems supports higher-order programs, and neither supports custom-programmed proof procedures, for cases where standard automation is insufficient.

The ESC/Java (Flanagan et al. 2002) and Spec# (Barnett et al. 2004) systems tackle some related problems within the classical verification framework. These systems have strictly less support for modeling data structures than Jahob has, so that it is impractical to use them to perform full verifications of many data structures.

A number of systems have been proposed recently to support dependently-typed programming in a setting oriented more towards traditional software development than Coq is. Agda (Norell 2007) and Epigram (McBride and McKinna 2004) are designed to increase the convenience of programming in type theory over what Coq provides, but, out of the box, these systems support neither imperative programming nor custom proof automation. ATS (Chen and Xi 2005) includes novel means for dealing with imperative state, but it includes no proof automation beyond decision procedures for simple base theories like linear arithmetic. This makes it much harder to write verified data structure implementations than in Ynot. Concoqtion (Pasalic et al. 2007) allows the use of Coq for reasoning about segments of general OCaml programs. While those programs may use imperativity, the Coq reasoning is restricted to pure index terms. Sage (Gronski et al. 2006) supports hybrid type-checking, where typing invariants may be specified with boolean-valued program functions and checked at runtime. This approach generally does not enable full static correctness verification.

Partly as a way to support imperative programming in type theory, Swierstra and Altenkirch (2007) have studied pure functional semantics for effectful programming language features, with embeddings in Haskell and Agda. Charguéraud and Pottier (2008) have demonstrated a translation from a calculus of capabilities to a pure functional language. In each case, the authors stated plans to do traditional interactive verification on the pure functional models that they generate. Since such verification is generally done in logics without general recursion, these translations cannot be used to verify general recursive programs without introducing an extra syntactic layer, in contrast to Ynot. Each other approach also introduces restrictions on the shape of the heap, such as the absence of stored impure functions in the case of Swierstra and Altenkirch's work.

Other computer proof assistants are based around pure functional programming languages, with opportunities for encoding and verifying imperative programs. Nonetheless, we see the elegance of our approach as depending on the confluence of a number of features not found in other mature proof assistants. ACL2 (Kaufmann and Moore 1997) does not support higher-order logic or higher-order functional programming. Bulwahn et al. (2008) describe a system for encoding and verifying impure monadic programs in Isabelle/HOL. Their implementation does not support storing functions in the heap. They suggest several avenues for loosening this restriction, and the approaches that support heap storage of impure functions involve restricting attention to functions that are constructive or continuous (properties that hold of all Coq functions), necessitating some extra proof burden or syntactic encoding.

There is closely related work in the field of shape analysis. The TVLA system (Sagiv et al. 2002) models heap shapes with a first-order logic with a built-in transitive closure operation. With the right choices of predicates that may appear in inferred specifications, TVLA is able to verify automatically many programs that involve both heap shape reasoning and reasoning in particular decidable theories such as arithmetic.

The Xisa system (Chang and Rival 2008) uses an approach similar to ours, as Xisa is based on user specification of inductive characterizations of shape invariants. Xisa builds this inductive definition mechanism into its framework, while we inherit a more general mechanism from Coq. Xisa is based on hardcoded algorithms for analyzing inductive definitions and determining when and how they should be unfolded. Such heuristics lack theoretical guarantees about how broadly they apply. In the design of our system, we recognize this barrier and allow users to extend the generic solver with custom rules for dealing with custom inductive predicates.

In comparing the new Ynot environment to the above systems and all others that we are aware of, there are a number of common advantages. No other system supports both highly-automated proofs based on separation logic (when they work) and highly human-guided proofs (when they are needed), let alone combinations of the two. None of the systems with significant automation support the combination of imperative and higher-order features, like we handle in the example of our higher-order memoizer and iterators. We also find no automated systems that deal with dependent types in programs. The first of these advantages seems critical in the verification of imperative programs that would be difficult to prove correct even if refactored to be purely functional. For instance, it seems plausible that our environment could be used eventually to build a verified compiler that uses imperative data structures for efficient dataflow analysis, unification in type inference, and so on. None of the purely-automated tools that we have surveyed could be applied to that purpose without drastic redesign. We are not aware of any previous toolkit for manual proof about imperative programs in proof assistants that would make the task manageable; the manual reasoning about state would overwhelm "the interesting parts" of compiler verification.

6. Conclusions & Future Work

The latest Ynot source distribution, including examples, can be downloaded from the project web site:

`http://ynot.cs.harvard.edu/`

Concurrency is a big area for future work on Ynot. Systems like Smallfoot (Berdine et al. 2005) do automated separation-logic reasoning about memory safety of concurrent programs. We would like to extend that work to full correctness verification, by designing a monadic version of concurrent separation logic that fits well within Coq.

The full potential of the Ynot approach also depends on explicit handling of other computational effects, such as exceptions and input-output. Our prior prototype handled the former, and ongoing work considers supporting the latter.

As with any project in automated theorem proving, there is always room for improvements to automation and inference. A future version of Ynot could benefit greatly in usability by incorporating abstract interpretation to infer specifications, as several automated separation logic tools already do.

Nonetheless, our current system already fills a crucial niche in the space of verification tools. We have presented the first tool that performs well empirically in allowing mixes of manual and highly-automated reasoning about heap-allocated data structures, as well as the first tool to provide aggressive automation in proofs of higher-order, imperative programs. We hope that this will form a significant step towards full functional verification of imperative programs with deep correctness theorems.

References

Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Proc. CASSIS*, 2004.

Bruno Barras and Bruno Bernardo. The Implicit Calculus of Constructions as a programming language with dependent types. In *Proc. FoSSaCS*, 2008.

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Proc. FMCO*, 2005.

Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkök, and John Matthews. Imperative functional programming with Isabelle/HOL. In *Proc. TPHOLs*, 2008.

Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *Proc. POPL*, 2008.

Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *Proc. ICFP*, 2008.

Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In *Proc. ICFP*, 2005.

David Delahaye. A tactic language for the system Coq. In *Proc. LPAR*, 2000.

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. PLDI*, 2002.

Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proc. POPL*, 2004.

Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. Sage: Hybrid checking for flexible specifications. In *Proc. Scheme Workshop*, 2006.

Matt Kaufmann and J. S. Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Trans. Softw. Eng.*, 23(4), 1997.

Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proc. POPL*, 2006.

Barbara Liskov and Stephen N. Zilles. Specification techniques for data abstractions. *IEEE Trans. Software Eng.*, 1(1):7–19, 1975.

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

Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in Hoare Type Theory. *Proc. ICFP*, 2006.

Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Reasoning with the awkward squad. In *Proc. ICFP*, 2008.

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

Emir Pasalic, Jeremy Siek, Walid Taha, and Seth Fogarty. Concoqtion: Indexed types now! In *Proc. PEPM*, 2007.

Rasmus L. Petersen, Lars Birkedal, Aleksandar Nanevski, and Greg Morrisett. A realizability model for impredicative Hoare Type Theory. In *Proc. ESOP*, 2008.

John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. LICS*, 2002.

Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24, 2002.

Wouter Swierstra and Thorsten Altenkirch. Beauty in the beast: A functional semantics for the awkward squad. In *Proc. Haskell Workshop*, 2007.

Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. In *Proc. PLDI*, 2008.