The Ynot Tutorial

Adam Chlipala

July 27, 2009

Chapter 1

Introduction

Ynot is a library for the Coq proof assistant. Besides supporting mathematical theorem proving, Coq natively supports general functional programming, as its logic is an ML-like programming language. To preserve logical consistency, Coq's language Gallina rules out non-termination and side effects. Ynot adds those features in a controlled way, so that programs may be impure, while proofs remain pure and logically meaningful. Ynot goes further, combining the new impure constructs with a Hoare-style logic for proving the correctness of programs, including support for reasoning in the style of separation logic.

The basic approach stands in direct analogy with the way in which imperative features were added to Haskell. Haskell's IO monad reifies imperative programs as data that may be constructed by pure programs. Purity and referential transparency are preserved, as some system outside the scope of the language is responsible for "running" IO values. The situation is the same in Ynot. We define an indexed monad of impure programs, via uninterpreted Coq axioms. Coq's extraction facility can be used to translate programs that use these axioms into OCaml, Haskell, or Scheme programs. In these languages, the axioms can be realized via standard implementations of IO-style monads.

The Ynot library is designed to support effective engineering of certified programs. We include tactics that are able to automate much of reasoning about mutable heaps. Within that general framework, the user can plug in his own domain-specific tactics.

We will present the basics of the Ynot library through a series of examples. We assume that the reader is already familiar with programming and proving in Coq. There are a number of possible sources for this backround knowledge, including this draft textbook by the author of this tutorial:

`http://adam.chlipala.net/cpdt/`

Chapter 2

Mutable Counters

Our first example is trivial, designed to introduce the main features of Ynot. We will implement imperative natural number counters. First, we import the Ynot library.

```
Require Import Ynot.
```

For this to work, the compiled Ynot modules must be in the module path. Those modules can be compiled simply by running `make` in the root directory of the Ynot distribution. In a batch build of a file like the one we are writing here, the path to the Ynot library may be specified like:

```
coqc -R /path/to/ynot/src Ynot Counter.v
```

For interactive Emacs development with Proof General, it is useful to add a variable setting like this in your `.emacs` file.

```
(custom-set-variables
 '(coq-prog-args '("-R" "/path/to/ynot/src" "Ynot"))
 )
```

With such a setting, you should be able to execute the Require Import line without complaint. Afterward, we open a notation scope, to enable use of concise notations for assertions about heaps.

```
Open Local Scope hprop_scope.
```

Next, we write a module signature that defines the ADT (abstract data type) of mutable counters.

```
Module Type COUNTER.
  Parameter t : Set.
  Parameter rep : t → nat → hprop.
```

A counter has type t. In ML, an abstract type within a module usually enforces proper usage implicitly, where the set of values that may be constructed is limited strategically

through the choice of which methods to export. With mutable data structures in Ynot, this regime will not generally be enough. Instead, we employ an additional design pattern of *representation predicates*, as illustrated by *rep* in this example. A representation predicate takes a value of the ADT as an argument, and it usually also takes one or more other values that stand for *a pure functional model* of the imperative value. The type *hprop* stands for predicates over heaps. Thus, for a counter c and its pure functional model n, *rep c n* stands for the set of heaps that are consistent with the assumption that c represents n.

We see these parameters in use in the type of the counter *new* operation.

```
Parameter new : Cmd __ (fun c : t ⇒ rep c 0).
```

This type uses the *Cmd* type family, our main parameterized monad. The two explicit arguments are a precondition and a postcondition for this method, in the tradition of Hoare logic. The precondition __ describes an empty heap, and the postcondition fun $c \Rightarrow$ *rep c* 0 says that, if method execution terminates with a counter c, then c represents 0 in the final heap.

The name *Cmd* alludes to the foundation of this type family in *separation logic*, following the *small footprint* approach to specification. The *new* method does not actually require that the heap be empty when the method is called. Rather, the pre- and postconditions only specify the method's effects on *the part of the heap that the method touches*. We will see later how this property can be put to use in verification.

The *free* method has a similar type, but it uses one new feature.

```
Parameter free : ∀ (c : t) (n : [nat]), Cmd (n ~~ rep c n) (fun _ : unit ⇒ __).
```

We write the type *nat* in brackets to indicate that that method argument is *computationally irrelevant*. That is, n is a so-called "ghost state" argument, used only to help us prove the correctness of this method. The compiled version of this program will not contain n. In the precondition of *free*, we use the notation n ~~ p, where p is an *hprop* that may mention spec variable n. Each spec variable that an assertion uses must be unpacked explicitly in this way.

The type of the method for reading a counter's value introduces two more new assertion constructs.

```
Parameter get : ∀ (c : t) (n : [nat]), Cmd (n ~~ rep c n)
  (fun n' ⇒ n ~~ rep c n * [n' = n]).
```

In the postcondition, we see a sub-assertion $[n' = n]$. This is a lifted pure proposition; the assertion is true whenever $n' = n$ and the heap is empty. We combine that pure assertion with *rep c n* using the *separating comjunction* *. An assertion $p * q$ is true for heap h whenever h can be split into two disjoint pieces *h1* and *h2*, such that *h1* satisfies p and *h2* satisfies q.

Now the type of the counter increment method should be unsurprising.

```
Parameter inc : ∀ (c : t) (n : [nat]), Cmd (n ~~ rep c n)
```

```
    (fun _ : unit ⇒ n ~~ rep c (S n)).
End COUNTER.
```

We can implement a module ascribing to this signature. Since the types of the methods include specifications, we know that any implementation is correct, up to the level of detail that we included in the signature.

```
Module Counter : COUNTER.
  Definition t := ptr.
  Definition rep (p : t) (n : nat) := p -> n.
```

We represent a counter with the *ptr* type. Unlike ML ref types, Ynot pointer types don't carry information on the types of data that they point to. Rather, we rely on typed points-to facts within assertions. We see an example in the definition of *rep*: a counter-pointer p represents a number n if p points to heap memory containing n.

Since our method types contain specifications, we will need to do some proving to define the methods. We define a simple tactic t that is able to dispatch all of the proof obligations we will encounter.

```
  Ltac t := unfold rep; sep fail idtac.
```

We replace uses of *rep* by unfolding its definition, and we call the *sep* tactic from the Ynot library. *sep* is intended as a separation logic version of Coq's *intuition* tactic, which simplifies formulas of constructive propositional logic. We have no theoretical completeness guarantees for *sep*, but usage patterns are roughly the same. Just as *intuition* takes an optional argument giving a tactic to apply in solving leaves of its proof search, *sep* takes two domain-specific tactics as arguments. In later examples, we will see more interesting choices for those tactics.

Before we begin programming, we open another scope, this time to let us write ML-like syntax for Ynot programs.

```
  Open Scope stsepi_scope.
```

We implement the *new* method by *declaring it as a proof search goal*, so that we can use tactics to discharge the obligations that we will generate.

```
  Definition new : Cmd __ (fun c ⇒ rep c 0).
    refine {{New 0}}; t.
  Qed.
```

The *refine* tactic is the foundation of our implementation. In general, *refine* takes a term with holes in it, solving the current goal and adding the types of the holes as subgoals. There are holes in the term we pass to *refine* in defining *new*, but they are hidden by the Ynot syntax extensions. We write double braces around a Ynot program to indicate simultaneous strengthening of the precondition and weakening of the postcondition. We chain our tactic t onto the *refine*, so that t is applied to discharge every subgoal.

It is instructive to see exactly which subgoals are being proved for us.

```
Definition new' : Cmd __ (fun c ⇒ rep c 0).
  refine {{New 0}}.
 2 subgoals

============================
 __ ==> ?504 * __

subgoal 2 is:
 ∀ v : ptr, ?504 * v -> 0 ==> rep v 0
```

The first subgoal corresponds to strengthening the precondition; we see an implication between our stated precondition and the precondition that Coq inferred. We are trying to write a function with precondition __, while Coq has figured out that our implementation could actually be given any precondition. That fact shows up in the form of the second assertion, which is a unification variable conjoined with the empty heap, which is logically equivalent to that unification variable alone.

The same unification appears to the left of an implication in the second subgoal, which comes from weakening the postcondition. Where v is the method return value, we must show that any heap with some unknown part and some part containing just a mapping of v to 0 can be described by the appropriate instance of *rep*. In our automated script above, the unification variable had already been determined to be __ by this point, so that this goal can be proved by reflexivity of implication.

For the rest of this and the other examples, we won't show the obligations that are generated. You will no doubt need to inspect such obligations in writing your own Ynot programs, so it may be useful to play with the proof scripts in this tutorial, to see which obligations are generated and experiment with manual means of discharging them.

```
  Abort.
```

The remaining method definitions are (perhaps surprisingly) quite straightforward. We use ML-style operators for working with pointers, writing prefix ! for reading and infix ::= for writing. We use Haskell-style ← notation for the monad "bind" operator.

```
Definition free : ∀ c n, Cmd (n ~~ rep c n) (fun _ : unit ⇒ __).
  intros; refine {{Free c}}; t.
Qed.

Definition get : ∀ c n, Cmd (n ~~ rep c n) (fun n' ⇒ n ~~ rep c n * [n' = n]).
  intros; refine {{!c}}; t.
Qed.

Definition inc : ∀ c n, Cmd (n ~~ rep c n) (fun _ : unit ⇒ n ~~ rep c (S n)).
  intros; refine (
    n' ← !c;
    {{c ::= S n'}}
  ); t.
```

Qed.
End *Counter*.

Chapter 3

Mutable Stacks

Our next example demonstrates one of the simplest imperative data structures that really deserves the name: polymorphic mutable stacks.

```
Require Import List.
Require Import Ynot.
Set Implicit Arguments.
Open Local Scope hprop_scope.

Module Type STACK.
  Parameter t : Set → Set.
  Parameter rep : ∀ T, t T → list T → hprop.
```

Compared to the *COUNTER* example, our new type t differs in being polymorphic in the type of data that we store. The representation predicate is parameterized similarly, and we set the convention that the functional model of a T stack is a T list.

The first three stack methods don't involve any new concepts.

```
  Parameter new : ∀ T : Set,
    Cmd __ (fun s : t T ⇒ rep s nil).
  Parameter free : ∀ (T : Set) (s : t T),
    Cmd (rep s nil) (fun _ : unit ⇒ __).

  Parameter push : ∀ (T : Set) (s : t T) (x : T) (ls : [list T]),
    Cmd (ls ~~ rep s ls) (fun _ : unit ⇒ ls ~~ rep s (x :: ls)).
```

The type of the *pop* method demonstrates two important patterns. First, we can use arbitrary Coq computation in calculating a precondition or postcondition. Our *pop* method returns an *option* T, which will be *None* when the stack is empty. We use Coq's standard match expression form to case-analyze this return value, returning a different assertion for each case. The *Some* case uses an *hprop* version of the standard existential quantifier.

```
  Parameter pop : ∀ (T : Set) (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.

Module Stack : STACK.
```

We use Coq's section mechanism to scope the type variable T over all of our definitions.

```
Section Stack.
  Variable T : Set.

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

We can use a recursive *hprop*-valued function to define what it means for a particular list to be represented in the heap, starting from a particular head pointer.

```
  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.
```

As in the *Counter* example, we represent a stack as an untyped pointer, and we rely on the *rep* predicate to enforce proper typing of associated heap cells.

```
  Definition stack := ptr.
  Definition rep q ls := Exists po :@ option ptr, q -> po * listRep ls po.
```

We define a tactic that will be useful for domain-specific goal simplification. In larger examples, such a tactic definition would probably be significantly longer. Here, we only need to ask Coq to try solving goals by showing that they are contradictory, because two equated values of a datatype are built from different constructors.

```
  Ltac simplr := try discriminate.
```

A key component of effective Ynot automation is the choice of appropriate domain-specific *unfolding lemmas*. Such a lemma characterizes how an application of a representation predicate may be decomposed, when something is known about the structure of the arguments. Our first simple unfolding lemma says that any functional list represented by a null pointer must be nil. *sep* can complete the proof after we begin with a case analysis on the list.

```
  Theorem listRep_None : ∀ ls, listRep ls None ==> [ls = nil].
```

```
  destruct ls; sep fail idtac.
Qed.
```

A slightly more complicated lemma characterizes the shape of a list represented by a non-null pointer. Here we put our *simplr* tactic to use as the second argument to *sep*. In general, the tactic given as that second argument is tried throughout proof search, in attempts to discharge goals that the separation logic simplifier can't prove alone.

```
Theorem listRep_Some : ∀ ls hd,
  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 simplr.
Qed.
```

With these lemmas available, we are ready to define a tactic that will be passed as the first argument to *sep*. The tactic in that position is used to simplify the goal before beginning the main proof search. The tactic *simp_prem* will perform that function for us.

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

The *simpl_IfNull* tactic comes from the Ynot library. It simplifies goal patterns that arise from a syntax extension that we will see shortly. The more interesting part of *simp_prem* is the call to *simpl_prem*, another tactic from the library. *simpl_prem t* looks for premises (sub-formulas on the left of implications) that can be simplified by the tactic *t*. Here, we try to simplify by applying either of our unfolding lemmas.

With these pieces in place, we define a final *t* solver tactic by dropping our two parameters into the version that we used for *Counter*.

```
Ltac t := unfold rep; sep simp_prem simplr.

Open Scope stsepi_scope.
```

The first three method definitions are quite simple and use no new concepts.

```
Definition new : Cmd __ (fun s ⇒ rep s nil).
  refine {{New (@None ptr)}}; t.
Qed.

Definition free : ∀ s, Cmd (rep s nil) (fun _ : unit ⇒ __).
  intros; refine {{Free s}}; t.
Qed.

Definition push : ∀ s x ls, Cmd (ls ~~ rep s ls) (fun _ : unit ⇒ ls ~~ rep s (x :: ls)).
  intros; refine (hd ← !s;
    nd ← New (Node x hd);
    {{s ::= Some nd}}
  ); t.
```

```
  Qed.
```

The definition of *pop* introduces the *IfNull* syntax extension. An expression *IfNull x Then e1 Else e2* expands to a test on whether the variable x of some *option* type is null. If x is *None*, then the result is *e1*. If x is *Some y*, then the result is *e2*, with all occurrences of x replaced by y.

```
  Definition pop : ∀ s ls,
    Cmd (ls ~~ rep s ls)
    (fun xo ⇒ ls ~~ match xo with
                    | None ⇒ [ls = nil] * rep s ls
                    | Some x ⇒ Exists ls' :@ list T, [ls = x :: ls']
                                 * rep s ls'
                  end).
    intros; refine (hd ← !s;
      IfNull hd Then
        {{Return None}}
      Else
        nd ← !hd;
        Free hd;;
        s ::= next nd;;
        {{Return (Some (data nd))}}); t.
  Qed.
End Stack.
```

Finally, since the signature makes the type t polymorphic, we define a trivial wrapper that discards the type paramter.

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

Chapter 4

Mutable Queues

Mutable queues take a bit more work to implement than mutable stacks do, because we need to consider the manipulation of linked lists at both their fronts and their backs.

```
Require Import List.
Require Import Ynot.
Set Implicit Arguments.
Open Local Scope hprop_scope.

Module Type QUEUE.
```

The first four components are identical to those from *STACK*.

```
  Parameter t : Set → Set.
  Parameter rep : ∀ T, t T → list T → hprop.

  Parameter new : ∀ T,
    Cmd __ (fun q : t T ⇒ rep q nil).
  Parameter free : ∀ T (q : t T),
    Cmd (rep q nil) (fun _ : unit ⇒ __).
```

The type of the *enqueue* method is a little more complicated than *push*'s type, since we model enqueueing as addition to the end of a list.

```
  Parameter enqueue : ∀ T (q : t T) (x : T) (ls : [list T]),
    Cmd (ls ~~ rep q ls) (fun _ : unit ⇒ ls ~~ rep q (ls ++ x :: nil)).
```

The specification for *dequeue* is equivalent to the stack *pop* spec; we write it differently with the structure of our correctness proofs in mind.

```
  Parameter dequeue : ∀ T (q : t T) (ls : [list T]),
    Cmd (ls ~~ rep q ls) (fun xo ⇒ ls ~~ match xo with
                                         | None ⇒ [ls = nil] * rep q ls
                                         | Some x ⇒
                                           match ls with
                                             | nil ⇒ [False]
                                             | x' :: ls' ⇒ [x' = x] * rep q ls'
```

```
                                                    end
                                                end).
```

End *QUEUE*.

Module *Queue* : *QUEUE*.
The implementation begins the same way as for *Stack*, declaring a type parameter and defining a type of singly-linked list nodes.

```
Section Queue.
  Variable T : Set.

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

To allow us to describe a list with special focus on its last element, we define an alternate list representation, this time parameterized on both head and tail nodes.

```
  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.
```

A queue itself is a pair of pointers to the first and last nodes in the list. When the queue is empty, each pointer will point to a null pointer.

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

We make a final auxiliary definition before *rep*, parameterizing it by the functional list and the values pointed to by the front and back pointers, rather than by the queue and the list.

```
  Definition rep' ls fr ba :=
    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.
```

Now it's easy to define *rep*. We quantify existentially over the values pointed to by the queue fields and appeal to *rep'*.

```
  Definition rep q ls := Exists fr :@ option ptr, Exists ba :@ option ptr,
    front q –> fr * back q –> ba * rep' ls fr ba.
```

We define a short simplification tactic that we will plug into *sep* as its second argument. We constructed this code by iterating interactively with the method definitions that follow. We won't discuss the details of *simplr* any further, but you can try commenting out parts of it to see what parts of later code fail.

```
Ltac simplr := repeat (try congruence;
  match goal with
    | [ x : option ptr ⊢ _ ] ⇒ destruct x
    | [ H : Some _ = Some _ ⊢ _ ] ⇒ injection H; clear H; intros; subst
    | [ H : nil = ?ls ++ _ :: nil ⊢ _ ] ⇒ destruct ls; discriminate
  end);
eauto.
```

We have an unfolding lemma analogous to the first one that we saw for *Stack*.

```
Lemma rep_nil : ∀ q,
  rep q nil ==> front q -> @None ptr * back q -> @None ptr.
  unfold rep; sep fail simplr.
Qed.
```

Another simple lemma gives a simplification rule for a queue whose back pointer is known to be non-null, corresponding to the second unfolding lemma from *Stack*.

```
Lemma rep'_Some2 : ∀ ls o1 ba,
  rep' ls o1 (Some ba) ==> Exists ls' :@ list T, Exists x :@ T, Exists fr :@ ptr,
    [ls = ls' ++ x :: nil] * [o1 = Some fr] * listRep ls' fr ba * ba -> Node x None.
  unfold rep'; sep fail simplr.
Qed.
```

One more unfolding lemma is critical to automating our proofs. We need to simplify cases where the front pointer is non-null. This is tricky because our definition of *listRep* is oriented towards decomposing the list from the back. As a prelude to our final lemma, we prove some facts about pure lists. These facts could profitably be added to a generic list library. Some of these lemmas may seem confusingly trivial; we prove them so that they may be used as *auto* hints.

```
Lemma app_nil_middle : ∀ (x1 x2 : T),
  x1 :: x2 :: nil = x1 :: nil ++ x2 :: nil.
  reflexivity.
Qed.

Lemma app_nil_middle' : ∀ (x1 x2 x3 : T) ls,
  x1 :: x2 :: ls ++ x3 :: nil = x1 :: (x2 :: ls) ++ x3 :: nil.
  reflexivity.
Qed.

Lemma list_cases : ∀ ls : list T,
  ls = nil
```

```
    ∨ (∃ x, ls = x :: nil)
    ∨ (∃ x1, ∃ ls', ∃ x2, ls = x1 :: ls' ++ x2 :: nil).
    Hint Immediate app_nil_middle app_nil_middle'.

    induction ls; simpl; firstorder; subst; eauto 6.
  Qed.

  Lemma app_inj_tail' : ∀ (x1 : T) ls' x2 v v0,
    x1 :: ls' ++ x2 :: nil = v ++ v0 :: nil
    → x1 :: ls' = v ∧ x2 = v0.
    intros; apply app_inj_tail; assumption.
  Qed.

  Implicit Arguments app_inj_tail' [x1 ls' x2 v v0].
```

Finally, we come to the unfolding lemma we need. Again, it has a completely automated proof. We will not go into detail on how we arrived at this proof script; design was again driven by iteration with the code to follow. You can try omitting parts of this script or breaking it into stages to get a better idea of what is going on here.

```
  Lemma rep'_Some1 : ∀ ls fr ba,
    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.
    Ltac list_simplr := repeat (simplr ||
      match goal with
        | [ ls' : list T ⊢ _ ] ⇒
          match goal with
            | [ ⊢ context[ ([_ :: _ = ls' ++ _ :: nil] * listRep ls' _ _) ] ] ⇒ destruct ls'
          end
        | [ ⊢ context[[nil = _ ++ _ :: _]] ] ⇒
          inhabiter; search_prem ltac:(apply himp_inj_prem); intro
        | [ ⊢ context[[_ :: _ ++ _ :: _ = _ ++ _ :: _]] ] ⇒
          inhabiter; search_prem ltac:(apply himp_inj_prem); intro
        | [ H : _ ⊢ _ ] ⇒ generalize (app_inj_tail' H); clear H; intuition; subst; simpl
      end).

    destruct ba; simpl; [ | sep fail idtac ];
      generalize (list_cases ls); intuition; subst;
        repeat match goal with
                 | [ H : ex _ ⊢ _ ] ⇒ destruct H
               end;
```

```
    sep fail list_simplr.
Qed.
```

Now we've passed the hairy part of the development, and everything that follows is quite direct. We define our premise simplification tactic in almost the same way as for *Stack*. The *idtac*; at the beginning of the tactic works around a strange limitation in which tactics Coq will allow to be passed as arguments.

```
Ltac simp_prem :=
  idtac;
  simpl_prem ltac:(apply rep_nil || apply rep'_Some1 || apply rep'_Some2).

Ltac t := unfold rep; simpl_IfNull; sep simp_prem simplr.

Open Scope stsepi_scope.

Definition new : Cmd __ (fun q ⇒ rep q nil).
  refine (fr ← New (@None ptr);
    ba ← New (@None ptr);
    {{Return (Queue fr ba)}}); t.
Qed.
```

To verify *free*, we need a trivial fact about *rep'*. We add that fact as a hint, so that it is applied automatically during *sep*'s proof search.

```
Lemma rep'_nil : __ ==> rep' nil None None.
  t.
Qed.

Hint Resolve rep'_nil.

Definition free : ∀ q, Cmd (rep q nil) (fun _ : unit ⇒ __).
  intros; refine (Free (front q);;
    {{Free (back q)}}); t.
Qed.
```

To verify *enqueue*, we need to make our sole use of induction, proving a lemma about the effect of adding a node to the end of a list.

```
Lemma push_listRep : ∀ ba x nd ls fr,
  ba -> Node x (Some nd) * listRep ls fr ba ==> listRep (ls ++ x :: nil) fr nd.
  Hint Resolve himp_comm_prem.

  induction ls; t.
Qed.

Hint Immediate push_listRep.
```

One more lemma is useful. It may be surprising that we feel the need to prove this lemma, since our *t* discharges it trivially. Like in so many other similar situations, we prove the lemma explicitly and add it as a hint to guide unification by *eauto*.

```
Lemma push_nil : ∀ (x : T) nd,
  __ ==> [x :: nil = nil ++ x :: nil] * listRep nil nd nd.
  t.
Qed.

Hint Immediate push_nil.

Definition enqueue : ∀ q x ls, Cmd (ls ~~ rep q ls)
  (fun _ : unit ⇒ ls ~~ rep q (ls ++ x :: nil)).
  intros; refine (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}}); t.
Qed.
```

The definition of *dequeue* is similar to that of *enqueue*, and this time we need no new lemmas.

```
Definition dequeue : ∀ q ls,
  Cmd (ls ~~ rep q ls) (fun xo ⇒ ls ~~ match xo with
                                       | None ⇒ [ls = nil] * rep q ls
                                       | Some x ⇒
                                         match ls with
                                           | nil ⇒ [False]
                                           | x' :: ls' ⇒ [x' = x] * rep q ls'
                                         end
                                     end).
  intros; refine (fr ← !front q;
    IfNull fr Then
      {{Return None}}
    Else
      nd ← !fr;
      Free fr;;
      front q ::= next nd;;
      IfNull next nd As nnd Then
        back q ::= @None ptr;;
        {{Return (Some (data nd))}}
      Else
        {{Return (Some (data nd))}}); t.
  Qed.
End Queue.
```

```
  Definition t (_ : Set) := queue.
End Queue.
```

Chapter 5

Compilation

In an ideal world, we would have a specialized compiler from Coq to native code. More than one research group is working towards that goal today. For now, we can compile Ynot developments to reasonably efficient binaries using Coq's extraction mechanism. Some examples distributed with Ynot show how to do this, via extraction to OCaml, the primary extraction language supported by Coq. OCaml's type system is much less expressive than Coq's, so some Coq programs will not be translated to valid OCaml programs, but Ynot developments that avoid Coq's more arcane features are likely to be handled properly.

The basic extraction mechanism already does most of the work for us. We just need to craft build infrastructures around it, including uses of Coq's extraction hints, to map particular Coq identifiers to particular OCaml identifiers. This is especially necessary to realize the primitive program constructs of Ynot. These constructs must remain as axioms in Coq, since it is impossible to write imperative programs directly. Nonetheless, we achieve a consistent final result by instructing Coq to extract them as particular impure OCaml functions. The Ynot distribution includes in the `src/ocaml/` directory such an implementation of the primitives that we need.

The "Hello World" example in `examples/hello-world/` demonstrates how to build a program that uses IO, and the `linked-list` example demonstrates use of an imperative linked list ADT. In this tutorial, we will not go into detail on how the build process works, since it involves no theoretically deep elements. You can re-use our Makefiles to produce new programs to be compiled via OCaml by copying the simple file structure used for `hello-world`.

- `HelloWorld.v`, the Ynot development
- `Extract.v`, the Coq code to generate an OCaml equivalent
- `Makefile`, which defines some example-specific variables and calls out to a shared Makefile
- `ocaml/main.ml`, which gives the OCaml code to run for the final program, calling the main entry point from the extracted development

In a directory set up like this, running `make` builds only the Coq parts, and running `make build` generates the final executable. The latter target produces a native code executable `main.native` and a bytecode version `main.byte`.