SOFTWARE FOUNDATIONS

VOLUME 3: VERIFIED FUNCTIONAL ALGORITHMS

DECIDE

PROGRAMMING WITH DECISION PROCEDURES

```
Set Warnings "-notation-overridden,-parsing".
Require Import Perm.
```

Using `reflect` to characterize decision procedures

Thus far in *Verified Functional Algorithms* we have been using

- propositions (`Prop`) such as `a<b` (which is Notation for `lt a b`)
- booleans (`bool`) such as `a<?b` (which is Notation for `ltb a b`).

```
Check Nat.lt. (* : nat -> nat -> Prop *)
Check Nat.ltb. (* : nat -> nat -> bool *)
```

The Perm chapter defined a tactic called `bdestruct` that does case analysis on (x <? y) while giving you hypotheses (above the line) of the form (x<y). This tactic is built using the `reflect` type and the `blt_reflect` theorem.

```
Print reflect.
(* Inductive reflect (P : Prop) : bool -> Set :=
    | ReflectT : P -> reflect P true
    | ReflectF : ~ P -> reflect P false  *)

Check blt_reflect. (* : forall x y, reflect (x<y) (x <? y) *)
```

The name `reflect` for this type is a reference to *computational reflection*, a technique in logic. One takes a logical formula, or proposition, or predicate, and designs a syntactic embedding of this formula as an "object value" in the logic. That is, *reflect* the formula back into the logic. Then one can design computations expressible inside the logic that manipulate these syntactic object values. Finally, one proves that the computations make transformations that are equivalent to derivations (or equivalences) in the logic.

The first use of computational reflection was by Goedel, in 1931: his syntactic embedding encoded formulas as natural numbers, a "Goedel numbering." The second and third uses of reflection were by Church and Turing, in 1936: they encoded (respectively) lambda-expressions and Turing machines.

In Coq it is easy to do reflection, because the Calculus of Inductive Constructions (CiC) has Inductive data types that can easily encode syntax trees. We could, for example, take some of our propositional operators such as `and`, `or`, and make an `Inductive` type that is an encoding of these, and build a computational reasoning system for boolean satisfiability.

But in this chapter I will show something much simpler. When reasoning about less-than comparisons on natural numbers, we have the advantage that `nat` already an inductive type; it is "pre-reflected," in some sense. (The same for `Z`, `list`, `bool`, etc.)

Now, let's examine how `reflect` expresses the coherence between `lt` and `ltb`. Suppose we have a value `v` whose type is `reflect (3<7) (3<?7)`. What is `v`? Either it is

- ReflectT `P` (3<?7), where `P` is a proof of `3<7`, and `3<?7` is `true`, or
- ReflectF `Q` (3<?7), where `Q` is a proof of `~(3<7)`, and `3<?7` is `false`.

In the case of `3,7`, we are well advised to use `ReflectT`, because (3<?7) cannot match the `false` required by `ReflectF`.

```
Goal (3<?7 = true). Proof. reflexivity. Qed.
```

So `v` cannot be `ReflectF Q (3<?7)` for any `Q`, because that would not type-check. Now, the next question: must there exist a value of type `reflect (3<7) (3<?7)` ? The answer is yes; that is the `blt_reflect` theorem. The result of `Check blt_reflect`, above, says that for any `x,y`, there does exist a value (blt_reflect x y) whose type is exactly `reflect (x<y)(x<?y)`. So let's look at that value! That is, examine what `H`, and `P`, and `Q` are equal to at "Case 1" and "Case 2":

```
Theorem three_less_seven_1: 3<7.
Proof.
assert (H := blt_reflect 3 7).
remember (3<?7) as b.
destruct H as [P|Q] eqn:?.
* (* Case 1: H = ReflectT (3<7) P *)
apply P.
* (* Case 2: H = ReflectF (3<7) Q *)
compute in Heqb.
inversion Heqb.
Qed.
```

Here is another proof that uses `inversion` instead of `destruct`. The `ReflectF` case is eliminated automatically by `inversion` because `3<?7` does not match `false`.

```
Theorem three_less_seven_2: 3<7.
Proof.
assert (H := blt_reflect 3 7).
inversion H as [P|Q].
apply P.
Qed.
```

The `reflect` inductive data type is a way of relating a *decision procedure* (a function from X to `bool`) with a predicate (a function from X to `Prop`). The convenience of `reflect`, in the verification of functional programs, is that we can do `destruct (blt_reflect a b)`, which relates `a<?b` (in the program) to the `a<b` (in the proof). That's just how the `bdestruct` tactic works; you can go back to `Perm.v` and examine how it is implemented in the `Ltac` tactic-definition language.

Using `sumbool` to Characterize Decision Procedures

```
Module ScratchPad.
```

An alternate way to characterize decision procedures, widely used in Coq, is via the inductive type `sumbool`.

Suppose `Q` is a proposition, that is, `Q: Prop`. We say `Q` is *decidable* if there is an algorithm for computing a proof of `Q` or `¬Q`. More generally, when `P` is a predicate (a function from some type `T` to `Prop`), we say `P` is decidable when `∀x:T, decidable(P)`.

We represent this concept in Coq by an inductive datatype:

```
Inductive sumbool (A B : Prop) : Set :=
 | left : A → sumbool A B
 | right : B → sumbool A B.
```

Let's consider `sumbool` applied to two propositions:

```
Definition t1 := sumbool (3<7) (3>2).
Lemma less37: 3<7. Proof. omega. Qed.
Lemma greater23: 3>2. Proof. omega. Qed.

Definition v1a: t1 := left (3<7) (3>2) less37.
Definition v1b: t1 := right (3<7) (3>2) greater23.
```

A value of type `sumbool (3<7) (3>2)` is either one of:

- `left` applied to a proof of (3<7), or
- `right` applied to a proof of (3>2).

Now let's consider:

```
Definition t₂ := sumbool (3<7) (2>3).
Definition v2a: t₂ := left (3<7) (2>3) less37.
```

A value of type `sumbool (3<7) (2>3)` is either one of:

- `left` applied to a proof of (3<7), or
- `right` applied to a proof of (2>3).

But since there are no proofs of 2>3, only `left` values (such as `v2a`) exist. That's OK.

`sumbool` is in the Coq standard library, where there is `Notation` for it: the expression `{A}+{B}` means `sumbool A B`.

```
Notation "{ A } + { B }" := (sumbool A B) : type_scope.
```

A very common use of `sumbool` is on a proposition and its negation. For example,

```
Definition t₄ := ∀ a b, {a<b}+{~(a<b)}.
```

That expression, `∀a b, {a<b}+{~(a<b)}`, says that for any natural numbers `a` and `b`, either `a<b` or `a≥b`. But it is *more* than that! Because `sumbool` is an Inductive type with two constructors `left` and `right`, then given the `{3<7}+{~(3<7)}` you can pattern-match on it and learn *constructively* which thing is true.

```
Definition v₃: {3<7}+{~(3<7)} := left _ _ less37.

Definition is_3_less_7: bool :=
 match v₃ with
  | left _ _ _ ⇒ true
  | right _ _ _ ⇒ false
 end.

Eval compute in is_3_less_7. (* = true : bool *)

Print t₄. (* = forall a b : nat, {a < b} + {~ a < b} *)
```

Suppose there existed a value `lt_dec` of type `t₄`. That would be a *decision procedure* for the less-than function on natural numbers. For any nats `a` and `b`, you could calculate `lt_dec a b`, which would be either `left ...` (if `a<b` was provable) or `right ...` (if `~(a<b)` was provable).

Let's go ahead and implement `lt_dec`. We can base it on the function `ltb: nat → nat → bool` which calculates whether `a` is less than `b`, as a boolean. We already have a theorem that this function on booleans is related to the proposition `a<b`; that theorem is called `blt_reflect`.

```
Check blt_reflect. (* : forall x y, reflect (x<y) (x<?y) *)
```

It's not too hard to use `blt_reflect` to define `lt_dec`

```
Definition lt_dec (a: nat) (b: nat) : {a<b}+{~(a<b)} :=
match blt_reflect a b with
| ReflectT _ P ⇒ left (a < b) (¬ a < b) P
```

```
| ReflectF _ Q ⇒ right (a < b) (¬ a < b) Q
end.
```

Another, equivalent way to define `lt_dec` is to use definition-by-tactic:

```
Definition lt_dec' (a: nat) (b: nat) : {a<b}+{~(a<b)}.
  destruct (blt_reflect a b) as [P|Q]. left. apply P. right.
apply Q.
Defined.

Print lt_dec.
Print lt_dec'.

Theorem lt_dec_equivalent: ∀ a b, lt_dec a b = lt_dec' a b.
Proof.
intros.
unfold lt_dec, lt_dec'.
reflexivity.
Qed.
```

Warning: these definitions of `lt_dec` are not as nice as the definition in the Coq standard library, because these are not fully computable. See the discussion below.

```
End ScratchPad.
```

`sumbool` in the Coq Standard Library

```
Module ScratchPad2.
Locate sumbool. (* Coq.Init.Specif.sumbool *)
Print sumbool.
```

The output of `Print sumbool` explains that the first two arguments of `left` and `right` are implicit. We use them as follows (notice that `left` has only one explicit argument `P`:

```
Definition lt_dec (a: nat) (b: nat) : {a<b}+{~(a<b)} :=
match blt_reflect a b with
| ReflectT _ P ⇒ left P
| ReflectF _ Q ⇒ right Q
end.

Definition le_dec (a: nat) (b: nat) : {a≤b}+{~(a≤b)} :=
match ble_reflect a b with
| ReflectT _ P ⇒ left P
| ReflectF _ Q ⇒ right Q
end.
```

Now, let's use `le_dec` directly in the implementation of insertion sort, without mentioning `ltb` at all.

```
Fixpoint insert (x:nat) (l: list nat) :=
  match l with
  | nil ⇒ x::nil
  | h::t ⇒ if le_dec x h then x::h::t else h :: insert x t
 end.
```

```
Fixpoint sort (l: list nat) : list nat :=
  match l with
  | nil ⇒ nil
  | h::t ⇒ insert h (sort t)
end.

Inductive sorted: list nat → Prop :=
| sorted_nil:
    sorted nil
| sorted_1: ∀ x,
    sorted (x::nil)
| sorted_cons: ∀ x y l,
   x ≤ y → sorted (y::l) → sorted (x::y::l).
```

Exercise: 2 stars (insert_sorted_le_dec)

```
Lemma insert_sorted:
  ∀ a l, sorted l → sorted (insert a l).
Proof.
  intros a l H.
  induction H.
  - constructor.
  - unfold insert.
    destruct (le_dec a x) as [ Hle | Hgt].
```

Look at the proof state now. In the first subgoal, we have above the line, `Hle: a ≤ x`. In the second subgoal, we have `Hgt: ¬ (a < x)`. These are put there automatically by the `destruct (le_dec a x)`. Now, the rest of the proof can proceed as it did in `Sort.v`, but using `destruct (le_dec _ _)` instead of `bdestruct (_ <=? _)`.

```
(* FILL IN HERE *) Admitted.
```

☐

Decidability and Computability

Before studying the rest of this chapter, it is helpful to study the `ProofObjects` chapter of *Software Foundations volume 1* if you have not done so already.

A predicate `P: T→Prop` is *decidable* if there is a computable function `f: T→bool` such that, forall `x:T`, `f x = true ↔ P x`. The second and most famous example of an *undecidable* predicate is the Halting Problem (Turing, 1936): `T` is the type of Turing-machine descriptions, and `P(x)` is, Turing machine `x` halts. The first, and not as famous, example is due to Church, 1936 (six months earlier): test whether a lambda-expression has a normal form. In 1936-37, as a first-year PhD student before beginning his PhD thesis work, Turing proved these two problems are equivalent.

Classical logic contains the axiom `∀P, P ∨ ¬P`. This is not provable in core Coq, that is, in the bare Calculus of Inductive Constructions. But its negation is not provable either. You could add this axiom to Coq and the system would still be consistent (i.e., no way to prove `False`).

But `P ∨ ¬P` is a weaker statement than `{P}+{~P}`, that is, `sumbool P (¬P)`. From `{P}+{~P}` you can actually *calculate* or `compute` either `left (x:P)` or `right(y: ¬P)`. From `P ∨ ¬P` you cannot `compute` whether `P` is true. Yes, you can `destruct` it in a proof, but not in a calculation.

For most purposes its unnecessary to add the axiom `P ∨ ¬P` to Coq, because for specific predicates there's a specific way to prove `P ∨ ¬P` as a theorem. For example, less-than on natural numbers is decidable, and the existence of `blt_reflect` or `lt_dec` (as a theorem, not as an axiom) is a demonstration of that.

Furthermore, in this "book" we are interested in *algorithms*. An axiom `P ∨ ¬P` does not give us an algorithm to compute whether P is true. As you saw in the definition of `insert` above, we can use `lt_dec` not only as a theorem that either `3<7` or `~(3<7)`, we can use it as a function to compute whether `3<7`. In Coq, you can't compute with axioms! Let's try it:

```
Axiom lt_dec_axiom_1: ∀ i j: nat, i<j ∨ ~(i<j).
```

Now, can we use this axiom to compute with?

```
(* Uncomment and try this:
Definition max (i j: nat) : nat :=
   if lt_dec_axiom_1 i j then j else i.
*)
```

That doesn't work, because an `if` statement requires an `Inductive` data type with exactly two constructors; but `lt_dec_axiom_1 i j` has type `i<j ∨ ~(i<j)`, which is not Inductive. But let's try a different axiom:

```
Axiom lt_dec_axiom_2: ∀ i j: nat, {i<j} + {~(i<j)}.

Definition max_with_axiom (i j: nat) : nat :=
   if lt_dec_axiom_2 i j then j else i.
```

This typechecks, because `lt_dec_axiom_2 i j` belongs to type `sumbool (i<j) (~(i<j))` (also written `{i<j} + {~(i<j)}`), which does have two constructors.

Now, let's use this function:

```
Eval compute in max_with_axiom 3 7.
  (*  = if lt_dec_axiom_2 3 7 then 7 else 3
     : nat *)
```

This `compute` didn't compute very much! Let's try to evaluate it using `unfold`:

```
Lemma prove_with_max_axiom: max_with_axiom 3 7 = 7.
Proof.
unfold max_with_axiom.
try reflexivity. (* does not do anything, reflexivity fails *)
(* uncomment this line and try it:
   unfold lt_dec_axiom_2.
*)
destruct (lt_dec_axiom_2 3 7).
reflexivity.
```

```
contradiction n. omega.
Qed.
```

It is dangerous to add Axioms to Coq: if you add one that's inconsistent, then it leads to the ability to prove `False`. While that's a convenient way to get a lot of things proved, it's unsound; the proofs are useless.

The Axioms above, `lt_dec_axiom_1` and `lt_dec_axiom_2`, are safe enough: they are consistent. But they don't help in computation. Axioms are not useful here.

```
End ScratchPad2.
```

Opacity of Qed

This lemma `prove_with_max_axiom` turned out to be *provable*, but the proof could not go by *computation*. In contrast, let's use `lt_dec`, which was built without any axioms:

```
Lemma compute_with_lt_dec: (if ScratchPad2.lt_dec 3 7 then 7
else 3) = 7.
Proof.
compute.
(* uncomment this line and try it:
   unfold blt_reflect.
*)
Abort.
```

Unfortunately, even though `blt_reflect` was proved without any axioms, it is an *opaque theorem* (proved with `Qed` instead of with `Defined`), and one cannot compute with opaque theorems. Not only that, but it is proved with other opaque theorems such as `iff_sym` and `Nat.ltb_lt`. If we want to compute with an implementation of `lt_dec` built from `blt_reflect`, then we will have to rebuild `blt_reflect` without using `Qed` anywhere, only `Defined`.

Instead, let's use the version of `lt_dec` from the Coq standard library, which *is* carefully built without any opaque (`Qed`) theorems.

```
Lemma compute_with_StdLib_lt_dec: (if lt_dec 3 7 then 7 else 3)
= 7.
Proof.
compute.
reflexivity.
Qed.
```

The Coq standard library has many decidability theorems. You can examine them by doing the following `Search` command. The results shown here are only for the subset of the library that's currently imported (by the `Import` commands above); there's even more out there.

```
Search ({_}+{~_}).
(*
```

```
reflect_dec: forall (P : Prop) (b : bool), reflect P b -
> {P} + {~ P}
lt_dec: forall n m : nat, {n < m} + {~ n < m}
list_eq_dec:
  forall A : Type,
  (forall x y : A, {x = y} + {x <> y}) ->
  forall l l' : list A, {l = l'} + {l <> l'}
le_dec: forall n m : nat, {n <= m} + {~ n <= m}
in_dec:
  forall A : Type,
  (forall x y : A, {x = y} + {x <> y}) ->
  forall (a : A) (l : list A), {In a l} + {~ In a l}
gt_dec: forall n m : nat, {n > m} + {~ n > m}
ge_dec: forall n m : nat, {n >= m} + {~ n >= m}
eq_nat_decide: forall n m : nat, {eq_nat n m} + {~ eq_nat n m}
eq_nat_dec: forall n m : nat, {n = m} + {n <> m}
bool_dec: forall b1 b2 : bool, {b1 = b2} + {b1 <> b2}
Zodd_dec: forall n : Z, {Zodd n} + {~ Zodd n}
Zeven_dec: forall n : Z, {Zeven n} + {~ Zeven n}
Z_zerop: forall x : Z, {x = 0%Z} + {x <> 0%Z}
Z_lt_dec: forall x y : Z, {(x < y)%Z} + {~ (x < y)%Z}
Z_le_dec: forall x y : Z, {(x <= y)%Z} + {~ (x <= y)%Z}
Z_gt_dec: forall x y : Z, {(x > y)%Z} + {~ (x > y)%Z}
Z_ge_dec: forall x y : Z, {(x >= y)%Z} + {~ (x >= y)%Z}
*)
```

The type of `list_eq_dec` is worth looking at. It says that if you have a decidable equality for an element type `A`, then `list_eq_dec` calculates for you a decidable equality for type `list A`. Try it out:

```
Definition list_nat_eq_dec:
    (∀ al bl : list nat, {al=bl}+{al≠bl}) :=
  list_eq_dec eq_nat_dec.

Eval compute in if list_nat_eq_dec [1;3;4] [1;4;3] then true
else false.
 (* = false : bool *)

Eval compute in if list_nat_eq_dec [1;3;4] [1;3;4] then true
else false.
 (* = true : bool *)
```

Exercise: 2 stars (list_nat_in)

Use `in_dec` to build this function.

```
Definition list_nat_in: ∀ (i: nat) (al: list nat), {In i al}+{~
In i al}
 (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.

Example in_4_pi: (if list_nat_in 4 [3;1;4;1;5;9;2;6] then true
else false) = true.
Proof.
simpl.
(* reflexivity. *)
(* FILL IN HERE *) Admitted.
```

□

In general, beyond `list_eq_dec` and `in_dec`, one can construct a whole programmable calculus of decidability, using the programs-as-proof language of Coq. But is it a good idea? Read on!

Advantages and Disadvantages of `reflect` Versus `sumbool`

I have shown two ways to program decision procedures in Coq, one using `reflect` and the other using `{_}+{~_}`, i.e., `sumbool`.

- With `sumbool`, you define *two* things: the operator in `Prop` such as `lt: nat → nat → Prop` and the decidability "theorem" in `sumbool`, such as `lt_dec: ∀i j, {lt i j}+{~ lt i j}`. I say "theorem" in quotes because it's not *just* a theorem, it's also a (nonopaque) computable function.
- With `reflect`, you define *three* things: the operator in `Prop`, the operator in `bool` (such as `ltb: nat → nat → bool`, and the theorem that relates them (such as `ltb_reflect`).

Defining three things seems like more work than defining two. But it may be easier and more efficient. Programming in `bool`, you may have more control over how your functions are implemented, you will have fewer difficult uses of dependent types, and you will run into fewer difficulties with opaque theorems.

However, among Coq programmers, `sumbool` seems to be more widely used, and it seems to have better support in the Coq standard library. So you may encounter it, and it is worth understanding what it does. Either of these two methods is a reasonable way of programming with proof.