Some notes and notations . . .

This `import` requires `src/Tactics.v` compiled:

```
Require Import Tactics.
Set Implicit Arguments.
```

Also getting `MoreSpecif` from CPDT:

```
coqc MoreSpecif.v
```

```
Require Import MoreSpecif.
Open Scope specif_scope.
```

`pred_string` with its *SubSet* type takes a theorem only as non-implicit argument. It uses:

```
Notation "!" := (False_rec _ _) : specif_scope.
Notation "[ e ]" := (exist _ e _) : specif_scope.
```

```
Definition pred_strong : ∀ n : nat, n > 0 → {m : nat | n = S m}.
  refine (fun n ⇒
    match n with
      | O ⇒ fun _ ⇒ !
      | S n' ⇒ fun _ ⇒ [n']
    end); crush.
Defined.
```

```
Theorem two_gt0 : 2 > 0.
  crush.
Qed.
```

```
Eval compute in pred_strong two_gt0.
```

With

```
Notation "'Yes'" := (left _ _).
Notation "'No'" := (right _ _).
Notation "'Reduce' x" := (if x then Yes else No) (at level 50).
```

`eq_nat_dec` compares two natural numbers, returning either a proof of their equality or a proof of their inequality:

```
Definition eq_nat_dec : ∀ n m : nat, {n = m} + {n ≠ m}.
  refine (fix f (n m : nat) : {n = m} + {n ≠ m} :=
    match n, m with
      | O, O ⇒ Yes
      | S n', S m' ⇒ Reduce (f n' m')
      | _, _ ⇒ No
```

```
    end); congruence.
Defined.
```

```
Eval compute in eq_nat_dec 2 2.
Eval compute in eq_nat_dec 2 3.
```

Using Coq's:

```
Inductive maybe (A : Set) (P : A -> Prop) : Set :=
    Unknown : maybe P | Found : forall x : A, P x -> maybe P
```

And Adam's:

```
Notation "{{ x | P }}" := (maybe (fun x => P)).
Notation "??" := (Unknown _).
Notation "[[ x ]]" := (Found _ x _).

Definition pred_strong_opt : ∀ n : nat, {{m | n = S m}}.
  refine (fun n ⇒
    match n with
      | O ⇒ ??
      | S n' ⇒ [[n']]
   end); trivial.
Defined.
Eval compute in pred_strong_opt 2.
Eval compute in pred_strong_opt 0.
```

Pseudo-Monadic notation: `Notation "x <- e1 ; e2"` (propagates the maybe).

```
Definition doublePred : ∀ n1 n2 : nat, {{p | n1 = S (fst p) ∧ n2 = S (snd p)}}.
  refine (fun n1 n2 ⇒
    m1 ← pred_strong_opt n1;
    m2 ← pred_strong_opt n2;
    [[(m1, m2)]]); tauto.
Defined.
```

`Notation "e1 ;; e2" := (if e1 then e2 else ??)` (maybe => ASSERT)

```
Definition positive_difference :
  ∀ n m : nat, {{ k | k ≥ 0 ∧ k = n - m }}.
  refine (fun n m ⇒
    Compare_dec.le_dec m n;;
    [[ n - m ]]); crush.
Defined.
Eval compute in (positive_difference 4 3).
Eval compute in (positive_difference 3 5).
Eval compute in (positive_difference 4 4).
```

The **sumor-based** type is maximally expressive; any implementation of the type has the same input-output behavior.

```
Inductive sumor (A : Type) (B : Prop) : Type :=
    inleft : A -> A + {B} | inright : B -> A + {B}
Notation "!!" := (inright _ _).
Notation "[[[ x ]]]" := (inleft _ [x]).
```

```
Definition pred_strong_sumor : ∀ n : nat, {m : nat | n = S m} + {n = 0}.
  refine (fun n ⇒
    match n with
      | O ⇒ !!
      | S n' ⇒ [[[n']]]
   end); trivial.
Defined.
```

```
Eval compute in pred_strong_sumor 2.
Eval compute in pred_strong_sumor 0.
```

```
Notation "x <-- e1 ; e2" := (match e1 with
                                | inright _ => !!
                                | inleft (exist x _) => e2
                              end)
(right associativity, at level 60).
```

```
Definition doublePred' : ∀ n1 n2 : nat,
  {p : nat × nat | n1 = S (fst p) ∧ n2 = S (snd p)} + {n1 = 0 ∨ n2 = 0}.
  refine (fun n1 n2 ⇒
    m1 <– pred_strong_sumor n1;
    m2 <– pred_strong_sumor n2;
    [[[(m1, m2)]]]); tauto.
Defined.
```

pseudo-monadic assertion with sumor:

```
Notation "e1 ;;; e2" := (if e1 then e2 else !!)
```

```
Definition positive_difference_or_proof_n_le_m:
  ∀ n m : nat, {k | k ≥ 0 ∧ k = n - m} + { n < m }.
refine (fun n m ⇒
  Compare_dec.le_dec m n;;;
  [[[ n - m ]]]);
crush.
Defined.
Eval compute in (positive_difference_or_proof_n_le_m 4 3).
Eval compute in (positive_difference_or_proof_n_le_m 3 5).
Eval compute in (positive_difference_or_proof_n_le_m 4 4).
```