hack.nbitstream

Bit notation for bool

```
Definition of_Z (z : Z) : option bool :=
  match z with
  | 0 => Some false
  | 1 => Some true
  | _ => None
  end%Z.

Definition to_Z (b : bool) : Z :=
  if b then 1%Z else 0%Z.

Goal forall x : bool, of_Z (to_Z x) = Some x.
  intros x; destruct x; compute; trivial.
Qed.

Number Notation bool of_Z to_Z : bit_scope.
```

Bit streams

We represent bit streams as binary natural numbers `N`. The stream is parameterized by the length.

```
Variant stream (sz : nat) := streamOf : N -> stream sz.
Arguments streamOf {sz}.
Definition empty : stream 0 := streamOf 0.
Definition single (b : bool) : stream 1 :=
  if b then streamOf 1 else streamOf 0.

Definition lowerBits sz n := N.land n (N.ones sz).
Definition to_N {sz} (str : stream sz) : N :=
  match str with
  | streamOf n => lowerBits (N.of_nat sz) n
  end.

Definition of_N {sz} : N -> stream sz := streamOf \o lowerBits (N.of_nat sz).

Definition normalise {sz} : stream sz -> stream sz := streamOf \o to_N.

Module eq.

  Definition rel {sz}(str1 str2 : stream sz) : Prop := to_N str1 = to_N str2.

  Lemma refl {sz}(str : stream sz) : rel str str.
    by rewrite /rel.
  Qed.

  Lemma symm {sz}(str1 str2 : stream sz) : rel str1 str2 -> rel str2 str1.
```

```
    rewrite /rel; by move => H; rewrite -H.
  Qed.

  Lemma trans {sz}(str1 str2 str3 : stream sz) : rel str1 str2 -> rel str2 str3 ->
rel str1 str3.
    rewrite /rel; by move => h1 h2; rewrite h1 h2.
  Qed.

End eq.

Infix "≡" := (eq.rel) (at level 70).

Add Parametric Relation {sz} : (stream sz) (@eq.rel sz)
    reflexivity proved by (@eq.refl sz)
    symmetry proved by (@eq.symm sz)
    transitivity proved by (@eq.trans sz) as stream_eq_rel.

Definition zeros {sz} : stream sz := streamOf 0%N.
Definition ones {sz} : stream sz := streamOf (N.ones (N.of_nat sz)).

Definition imap {A} (f : N -> A) {sz} : stream sz -> A := f \o to_N.

Definition transform (f : N -> N) {sz1 sz2} : stream sz1 -> stream sz2 := streamOf \o
imap f.
Definition map (f : N -> N) {sz} : stream sz -> stream sz := transform f.
Definition transform2 (f : N -> N -> N) {sz1 sz2 sz3} (sa : stream sz1) : stream sz2
-> stream sz3 :=
  transform (imap f sa).
Definition map2 (f : N -> N -> N) {sz} : stream sz -> stream sz -> stream sz :=
transform2 f.
```

Appending and splitting

```
Definition append {sz1 sz2} : stream sz1 -> stream sz2 -> stream (sz1 + sz2) :=
  transform2 (fun x y => N.lor (N.shiftl x (N.of_nat sz2)) y).

Definition take sz1 {sz2} : stream (sz1 + sz2) -> stream sz1 :=
  transform (fun x => N.shiftr x (N.of_nat sz2)).
Definition drop sz1 {sz2} : stream (sz1 + sz2) -> stream sz2 :=
  normalise \o transform id.

Definition split {sz1 sz2} (str : stream (sz1 + sz2)) : stream sz1 * stream sz2 :=
(take sz1 str, drop sz1 str).
```

List like cons

```
Definition cons (b : bool){sz} : stream sz -> stream (S sz) :=
  transform (fun n => if b then N.setbit n (N.of_nat sz) else n).

Definition head {sz} : stream (S sz) -> bool :=
  imap (fun n => N.testbit n (N.of_nat sz)).

Definition tail {sz} : stream (S sz) -> stream sz :=
  transform id.
```

```
Fixpoint vector {n}(vec : Vector.t bool n) : stream n :=
  match vec with
  | [] => empty
  | (x :: xs) => cons x (vector xs)
  end.

Infix "++" := append : bit_scope.
Infix "::" := cons : bit_scope.
```

Textual Encoding

We give three different ways to get the textual representation in base2, in hexadecimal and as bytes.

```
Require Import String.

Definition chunk (sz : N) (state : list N * N) :=
  let (chunks, n) := state in
  (lowerBits sz n :: chunks , N.shiftr n sz)%list.

Definition chunks (base: N) (sz : N) (n : N) : list N := fst (N.iter (sz/base) (chunk
base) (List.nil , n)).
Definition base2n (base : N) {sz} : stream sz -> list N := imap (chunks base
(N.of_nat sz)).

Require Import String.
Require Import Ascii.
Import AsciiSyntax.

Definition of_bin (i : N) : ascii :=
  (if (i =? 0)%N then "0" else "1")%char.

Definition of_hex (i : N) : ascii :=
  let lt10 := ascii_of_N (N_of_ascii "0" + i) in
  let ge10 := ascii_of_N (N_of_ascii "A" + (i - 10) ) in
  if (i <? 10)%N then lt10 else ge10.
```

Encode in arbitrary base

```
Definition encode base (enc : N -> ascii) {sz} : stream sz -> string :=
  string_of_list_ascii \o List.map enc \o base2n base.
```

Encode in base 2

```
Definition base2 {sz} := @encode 1 (of_bin) sz.
```

Encode in hexadecimal

```
Definition hex {sz} := @encode 4 (of_hex) sz.
```

Encode as bytes

```
Definition bytes {sz} := @encode 8 ascii_of_N sz.

Local Open Scope bit_scope.
```

Proof of correctness

```
Require Import ssreflect.

Lemma pow2_n_sn (n : N) : (2^n mod 2^(N.succ n) = 2^n)%N.
  rewrite N.mod_small; by [try (apply N.pow_lt_mono_r; lia)].
Qed.

Lemma mod_pow2_n_sn (n sz : N) : ((n mod 2 ^ sz) mod 2^(N.succ sz) = n mod 2^sz)%N.
  apply N.mod_small; transitivity (2^sz)%N.
  - by apply N.mod_upper_bound; apply N.pow_nonzero.
  - apply N.pow_lt_mono_r; lia.
Qed.

Lemma head_cons b {sz} (str : stream sz) : head (b :: str) = b.
  destruct str; destruct b; rewrite /head /cons /transform /imap /to_N /lowerBits /
comp.
  - rewrite N.setbit_spec' N.land_lor_distr_l ?N.land_ones Nnat.Nat2N.inj_succ
pow2_n_sn N.lor_spec.
    apply /orP; right.
    destruct sz.
    * rewrite //=.
    * by apply N.pow2_bits_true.
  - rewrite ?N.land_ones Nnat.Nat2N.inj_succ mod_pow2_n_sn N.mod_pow2_bits_high;
reflexivity.
Qed.

Lemma tail_cons b {sz} (str : stream sz) : tail (b :: str) ≡ str.
  destruct str; destruct b;
    by rewrite /tail /cons /transform /imap /eq.rel /to_N /lowerBits /comp
         ?N.setbit_spec' ?N.land_lor_distr_l
         ?N.land_ones Nnat.Nat2N.inj_succ mod_pow2_n_sn ?pow2_n_sn
         N.Div0.mod_mod ?N.Div0.mod_same ?N.lor_0_r.
Qed.
```

Index
