```
Require Import List.
Require Import String.
Require Import ZArith.

Open Scope list_scope.
Open Scope string_scope.
Open Scope Z_scope.

Require Import ImpSyntax.

(** Tests *)

Definition pred_of_dec {A B}
  (f : forall a1 a2, {B a1 a2} + {~ B a1 a2})
  (a1 a2 : A) : bool :=
  if f a1 a2 then true else false.

Definition val_eq_dec :
  forall v1 v2 : val,
    {v1 = v2} + {v1 <> v2}.
Proof.
  decide equality.
  - apply Bool.bool_dec.
  - apply Z.eq_dec.
  - apply String.string_dec.
  - apply Z.eq_dec.
Defined.

Definition imp_eq := pred_of_dec val_eq_dec.
Definition imp_lt := pred_of_dec Z_lt_dec.
Definition imp_le := pred_of_dec Z_le_dec.

(** Stores *)

Definition store_0 : store :=
  nil.

Fixpoint update (s : store)
  (x : string) (v : val) : store :=
  match s with
  | nil => (x, v) :: nil
  | (x', v') :: rest =>
      if string_dec x x' then
        (x, v) :: rest
      else
        (x', v') :: update rest x v
  end.

Fixpoint updates (s : store)
  (xs : list string) (vs : list val) : option store :=
  match xs, vs with
  | nil, nil =>
      Some s
  | x :: xs', v :: vs' =>
      updates (update s x v) xs' vs'
  | _, _ => None
  end.

Fixpoint lkup (s : store)
  (x : string) : option val :=
  match s with
  | nil => None
  | (x', v) :: rest =>
      if string_dec x x' then
        Some v
      else
        lkup rest x
  end.
```



```
(** Heaps *)

Definition heap_0 : heap :=
  nil.

Fixpoint copy (n : nat) (v : val) : list val :=
  match n with
  | O => nil
  | S m => v :: copy m v
  end.

Definition alloc (h : heap)
  (i : Z) (v : val) : heap :=
  List.app h (Vint i :: copy (Z.to_nat i) v).

Fixpoint read (h : heap)
  (i : Z) : option val :=
  match h, i with
  | nil, _ => None
  | v :: vs, 0 => Some v
  | v :: vs, _ => read vs (Zpred i)
  end.

Fixpoint write (h : heap)
  (i : Z) (v : val) : option heap :=
  match h, i with
  | nil, _ => None
  | x :: xs, 0 => Some (v :: xs)
  | x :: xs, _ =>
      match write xs (Zpred i) v with
      | Some h' => Some (x :: h')
      | None => None
      end
  end.

(** Environments *)

Definition env : Type :=
  list func.

Fixpoint locate (e : env)
  (x : string) : option func :=
  match e with
  | nil => None
  | f :: rest =>
      match f with Func x' _ _ _ =>
        if string_dec x x' then
          Some f
        else
          locate rest x
      end
  end.

(** External Calls *)

Inductive extcall_spec :
  string -> list val -> heap ->
  val -> heap -> Prop :=
| print_spec :
    forall v h v',
      extcall_spec
        "print_val" (v :: nil) h
        v' h
| flush_spec :
    forall h v',
      extcall_spec
        "flush" nil h
        v' h
| sleep_spec :
```

```
    forall i h v',
      extcall_spec
        "sleep" (Vint i :: nil) h
        v' h
| read_bool_spec :
    forall h b,
      extcall_spec
        "read_bool" nil h
        (Vbool b) h
| read_int_spec :
    forall h i,
      extcall_spec
        "read_int" nil h
        (Vint i) h
| read_str_spec :
    forall h cs,
      extcall_spec
        "read_str" nil h
        (Vstr cs) h.

Definition extcall_args_ok
  (f : string) (vs : list val) (h : heap) : bool :=
  match f, vs with
  | "print_val", v :: nil => true
  | "flush", nil => true
  | "sleep", Vint i :: nil => true
  | "read_bool", nil => true
  | "read_int", nil => true
  | "read_str", nil => true
  | _, _ => false
  end.

Axiom extcall :
  string -> list val -> heap ->
  val * heap.

Axiom extcall_ok :
  forall f vs h v' h',
    extcall f vs h = (v', h') ->
    extcall_spec f vs h v' h'.

(** Common *)

Fixpoint zlen (h : heap) : Z :=
  match h with
  | nil => 0
  | _ :: rest => 1 + zlen rest
  end.

Definition isNop (p : stmt) :
  {p = Snop} + {p <> Snop}.
Proof.
  destruct p; auto;
    right; congruence.
Qed.

Definition store_eq_dec (s1 s2 : store) :
  {s1 = s2} + {s1 <> s2}.
Proof.
  decide equality.
  decide equality.
  - apply val_eq_dec.
  - apply string_dec.
Qed.

Definition expr_eq_dec (e1 e2 : expr) :
  {e1 = e2} + {e1 <> e2}.
Proof.
  decide equality.
```



```
  - apply val_eq_dec.
  - apply string_dec.
  - decide equality.
  - decide equality.
Qed.

Fixpoint size_s (p : stmt) : nat :=
  match p with
  | Snop
  | Sset _ _
  | Salloc _ _ _
  | Swrite _ _ _
  | Scall _ _ _ => O
  | Sifelse _ p1 p2 => S (size_s p1 + size_s p2)
  | Swhile _ p1 => S (size_s p1)
  | Sseq p1 p2 => S (size_s p1 + size_s p2)
  | Sincall p1 _ _ _ => S (size_s p1)
  end%nat.

Inductive result : Type :=
| Done : heap -> val -> result
| Timeout : store -> heap -> stmt -> expr -> result
| Stuck : store -> heap -> stmt -> expr -> result.
```