Mechanised Featherweight VeriFast – Machine-Readable Executable Definition and Machine-Checked Soundness Proof – The Coq Sources

Frédéric Vogels, Bart Jacobs, and Frank Piessens

November 6, 2013

Contents

Chapter 1

Library Util

```
Require Import String.
Require FMapWeakList.
Require Equalities.
Notation "x |> f" := (f x) (at level 90, only parsing).
Module StringMDT <: Equalities.MiniDecidableType.
  Definition t := string.
  Definition eq_dec := string_dec.
End StringMDT.
Module StringDT := Equalities.Make_UDT StringMDT.
Module StringMap := FMapWeakList.Make StringDT.
```

Chapter 2

Library Programs

```
Require Export ZArith.
Require Export Util.
Require Import String.
Require Export List.
Require ListSet.
```

```
Open Scope Z_scope.
```

```
Notation "[ ]" := nil.
Notation "[ x ]" := (cons x nil).
Notation "[ x , .. , y ]" := (cons x .. (cons y nil) ..).
```

2.1 Syntax of programs

2.1.1 Expressions

```
Notation var := string (only parsing).
Notation value := Z (only parsing).
```

```
Inductive expr :=
| e_lit(z: Z)
| e_var(x: var)
| e_add(e1 e2: expr)
.
```

Notations

```
Delimit Scope expr_scope with expr.
Coercion e_lit: Z >-> expr.
Coercion e_var: var >-> expr.
Infix "+" := e_add (at level 50, left associativity) : expr_scope.
```

2.1.2 Boolean expressions

```
Inductive bexpr :=
| b_eq(e1 e2: expr)
| b_lt(e1 e2: expr)
| b_not(b: bexpr)
.
```

Notations

```
Delimit Scope bexpr_scope with bexpr.
Infix "==" := b_eq (at level 70) : bexpr_scope.
Infix "<" := b_lt (at level 70): bexpr_scope.
Notation "~ b" := (b_not b) : bexpr_scope.
Definition b_true := (0 == 0)%bexpr.
```

2.1.3 Assertions

```
Definition pred := string.
Definition pointsto_pred := "|->"%string.
Definition malloc_block_pred := "mb"%string.
Inductive asn :=
  BAsn(b: bexpr)
| PAsn(p: pred)(es: list expr)(xs: list var)
| IfAsn(b: bexpr)(a1 a2: asn)
| SepAsn(a1 a2: asn)
.
Definition PointsTo l v := PAsn pointsto_pred [l] [v].
```

Notations

```
Delimit Scope asn_scope with asn.
Coercion BAsn: bexpr >-> asn.
Notation "x |-> ?? v" := (PointsTo x v) (at level 19, v at level 1) : asn_scope.
Notation "x |-> '_'" := (PointsTo x "_") (at level 19) : asn_scope.
Notation "'If' b 'Then' a1 'Else' a2" := (IfAsn b a1 a2) (at level 20) : asn_scope.
Infix "&*&" := SepAsn (at level 30, right associativity) : asn_scope.
Inductive pat :=
  LitPat(e: expr)
| VarPat(x: var)
.
```

```
Coercion LitPat: expr >-> pat.
Definition p_lit z := LitPat (e_lit z).
Definition p_var x := LitPat (e_var x).
Coercion p_lit: Z >-> pat.
Coercion p_var: var >-> pat.
Infix "+" := e_add (at level 50, left associativity) : pat_scope.

Notation "?? x" := (VarPat x) (at level 20) : pat_scope.
Notation " '??_' " := (VarPat "_") (at level 20) : pat_scope.

Fixpoint map_rem{A B}(f: A → option B)(xs: list A): list B × list A :=
  match xs with
    nil ⇒ (nil, nil)
  | x::xs' ⇒
    match f x with
      None ⇒ (nil, xs)
    | Some y ⇒
      let (ys, zs) := map_rem f xs' in
      (y::ys, zs)
    end
  end.

Definition PAsn'(p: pred)(ps: list pat): asn :=
  let (es, ps) := map_rem (fun p ⇒ match p with LitPat e ⇒ Some e | _ ⇒ None end) ps
in
  let (xs, ps) := map_rem (fun p ⇒ match p with VarPat x ⇒ Some x | _ ⇒ None end) ps
in
  match ps with
    nil ⇒ PAsn p es xs
  | _ ⇒ PAsn "<PAsn': syntax error>"%string [] []
  end.

Notation "# p ()" := (PAsn p [] []) (at level 1, p at level 5) : asn_scope.
Notation "# p ( )" := (PAsn p [] []) (at level 1, p at level 5) : asn_scope.
Definition cons_pat(p: pat)(ps: list pat) := p::ps.
Notation "# p ( p1 , .. , pn )" := (PAsn' p (cons_pat p1 .. (cons_pat pn nil) ..)) (at level
1, p at level 5) : asn_scope.
```

2.1.4 Commands

```
Definition routine := string.

Inductive cmd :=
| Assign(x: var)(e: expr)
| Malloc(x: var)(n: nat)
| Free(e: expr)
```

```
| Read(x: var)(e: expr)
| Write(e1 e2: expr)
| Call(x: var)(r: routine)(es: list expr)
| IfCmd(b: bexpr)(c1 c2: cmd)
| While(b: bexpr)(a: asn)(c: cmd)
| Seq(c1 c2: cmd)
| Open(p: pred)(es: list expr)
| Close(p: pred)(es: list expr)
| Skip
| Message(msg: string)
.
```

Notations

```
Delimit Scope cmd_scope with cmd.
Notation "x ":=" 'malloc' ( n )" := (Malloc x n) (at level 1) : cmd_scope.
Notation "'free' ( e )" := (Free e) (at level 1) : cmd_scope.
Notation "x ":=" [ e ]" := (Read x e) (at level 1) : cmd_scope.
Notation "x ":=" e" := (Assign x e) (at level 1) : cmd_scope.
Notation "[ e1 ] ":=" e2" := (Write e1 e2) (at level 0) : cmd_scope.
Notation "'If' b 'Then' c1 'Else' c2" := (IfCmd b c1 c2) (at level 20) : cmd_scope.
Notation "'while' b 'invariant' a 'do' c" := (While b a c) (at level 20) : cmd_scope.
Infix ";" := Seq (at level 30, right associativity) : cmd_scope.
Definition dummy_var := "_"%string.
Notation "'call' r ()" := (Call dummy_var r nil) (at level 1, r at level 5) : cmd_scope.
Notation "'call' r ( )" := (Call dummy_var r nil) (at level 1, r at level 5) : cmd_scope.
Definition cons_expr(e: expr) es := e::es.
Notation "'call' r ( e1 , .. , e2 )" := (Call dummy_var r (cons_expr e1 .. (cons_expr e2 nil)
..)) (at level 1, r at level 5) : cmd_scope.
Notation "x ":=" 'call' r ()" := (Call x r nil) (at level 1, r at level 5) : cmd_scope.
Notation "x ":=" 'call' r ( )" := (Call x r nil) (at level 1, r at level 5) : cmd_scope.
Notation "x ":=" 'call' r ( e1 , .. , e2 )" := (Call x r (cons_expr e1 .. (cons_expr e2 nil) ..))
(at level 1, r at level 5) : cmd_scope.
Notation "'open' p ()" := (Open p nil) (at level 1, p at level 5) : cmd_scope.
Notation "'open' p ( )" := (Open p nil) (at level 1, p at level 5) : cmd_scope.
Definition Open'(p: pred)(ps: list pat): cmd :=
  let (es, ps) := map_rem (fun p ⇒ match p with LitPat e ⇒ Some e | _ ⇒ None end) ps
in
  let (_, ps) := map_rem (fun p ⇒ match p with VarPat x ⇒ if string_dec x "_"%string
then Some tt else None | _ ⇒ None end) ps in
  match ps with
    nil ⇒ Open p es
  | _ ⇒ Open "<Open': syntax error>"%string []
```

```
  end.
Notation "'open' p ( p1 , .. , pn )" := (Open' p (cons_pat p1 .. (cons_pat pn nil) .. )) (at
level 1, p at level 5) : cmd_scope.
Notation "'close' p ()" := (Close p nil) (at level 1, p at level 5) : cmd_scope.
Notation "'close' p ( )" := (Close p nil) (at level 1, p at level 5) : cmd_scope.
Notation "'close' p ( e1 , .. , en )" := (Close p (cons_expr e1 .. (cons_expr en nil) ..)) (at
level 1, p at level 5) : cmd_scope.
```

2.1.5 Auxiliary definitions

```
Notation "xs 'Un' ys" := (ListSet.set_union string_dec xs ys) (at level 51, right associativity)
: list_set_scope.
Delimit Scope list_set_scope with list_set.

Fixpoint targets(c: cmd): list var :=
  match c with
     Assign x e ⇒ [x]
  | Malloc x n ⇒ [x]
  | Free e ⇒ []
  | Read x e ⇒ [x]
  | Write e1 e2 ⇒ []
  | Call x r es ⇒ [x]
  | IfCmd b c1 c2 ⇒ (targets c1 Un targets c2)%list_set
  | While b a c ⇒ targets c
  | Seq c1 c2 ⇒ (targets c1 Un targets c2)%list_set
  | Open p es ⇒ []
  | Close p es ⇒ []
  | Skip ⇒ []
  | Message m ⇒ []
  end.

Definition result: var := "result"%string.

Inductive pred_def := PredDef(xs: list var)(a: asn).
Definition pred_table := pred → option pred_def.

Inductive routine_def := RoutineDef(xs: list var)(c: cmd).
Definition routine_table := routine → option routine_def.

Inductive routine_spec := RoutineSpec(xs: list var)(pre post: asn).
Definition spec_table := StringMap.t routine_spec.
```

2.2 Semantics of expressions (shared between concrete and semiconcrete execution)

```
Definition store := var → value.

Definition store0: store := fun x ⇒ 0.
Definition store_update (s: store) x v x' := if string_dec x x' then v else s x'.

Fixpoint store_updates (s: store) xs vs :=
  match xs, vs with
    x::xs, v::vs ⇒ store_updates (store_update s x v) xs vs
  | _, _ ⇒ s
  end.

Fixpoint eval(s: store)(e: expr) :=
  match e with
    e_lit z ⇒ z
  | e_var x ⇒ s x
  | e_add e1 e2 ⇒ eval s e1 + eval s e2
  end.

Definition Z_eqb z1 z2 := if Z_eq_dec z1 z2 then true else false.
Definition Z_ltb z1 z2 := if Z_lt_dec z1 z2 then true else false.

Fixpoint beval(s: store)(b: bexpr): bool :=
  match b with
    b_eq e1 e2 ⇒ Z_eqb (eval s e1) (eval s e2)
  | b_lt e1 e2 ⇒ Z_ltb (eval s e1) (eval s e2)
  | b_not b ⇒ negb (beval s b)
  end.
```

Chapter 3

Library Outcomes

```
Require Export FunctionalExtensionality.
Require ClassicalEpsilon.
Require Setoid. Require Relation_Definitions.
Require Export ZArith.
Require Import String.
Require Export List.
Set Implicit Arguments.
```

3.1 Partially explorable sets

```
Inductive type_name :=
| n_Empty_set
| n_bool
| n_Z
| n_T(T: Type)
.
Fixpoint ltype_name(n: type_name): Type :=
  match n with
  | n_Empty_set ⇒ Empty_set
  | n_bool ⇒ bool
  | n_Z ⇒ Z
  | n_T T ⇒ T
  end.
Inductive set(X: Type) :=
| set_(n: type_name)(o: ltype_name n → X).
Definition empty_set {X} := set_ n_Empty_set (fun i ⇒ match i return X with end).
Definition pair_set {X} (x y: X) := set_ n_bool (fun b ⇒ if b then x else y).
Definition Z_set := set_ n_Z id.
```

```
Definition T_set T := set_ (n_T T) (fun x ⇒ x).
Definition set_comp {T X} (f: T → X): set X := set_ (n_T T) f.
Definition forall_{X}(s: set X)(P: X → Prop) :=
  match s with
    set_ n f ⇒ ∀ i, P (f i)
  end.
Notation "'forall'' x 'in' s , P" := (forall_ s (fun x ⇒ P)) (at level 200).
Definition exists_{X}(s: set X)(P: X → Prop) :=
  match s with
    set_ n f ⇒ ∃ i, P (f i)
  end.
Notation "'exists'' x 'in' s , P" := (exists_ s (fun x ⇒ P)) (at level 200).
Definition forall_intro{X}{P: X → Prop}(s: set X)(f: ∀ x, P x): forall_ s P :=
  match s with
    set_ n g ⇒ fun i ⇒ f (g i)
  end.
Definition set_img{X Y}(s: set X)(f: X → Y): set Y :=
  match s with
    set_ n g ⇒ set_ n (fun i ⇒ f (g i))
  end.
Local Notation "{ e | x 'in' s }" := (set_img s (fun x ⇒ e)).
```

3.2 Outcomes

```
Inductive outcome(S A: Type) :=
| single(s: S)(a: A)
| demonic(os: set (outcome S A))
| angelic(os: set (outcome S A))
| message(msg: string)(o: outcome S A)
.
Implicit Arguments single [S A].
```

3.3 Some shorthands

```
Definition oblock{S}{A}: outcome S A :=
  demonic empty_set.
Definition ofork{S}{A}(o1 o2: outcome S A) :=
  demonic (pair_set o1 o2).
```

```
Definition demonicT {I S A} (o: I → outcome S A): outcome S A :=
  demonic (set_comp o).
Definition ofail{S}{A}: outcome S A :=
  angelic empty_set.
Definition angelicT {I S A} (o: I → outcome S A): outcome S A :=
  angelic (set_comp o).
```

3.4 Semantics of outcomes

sat o Q means outcome *o* satisfies postcondition *Q*.

```
Fixpoint sat (S A: Type)(o: outcome S A)(Q: S → A → Prop): Prop :=
  match o with
    single s a ⇒ Q s a
  | demonic os ⇒ forall' o in os, sat o Q
  | angelic os ⇒ exists' o in os, sat o Q
  | message msg o ⇒ sat o Q
  end.
Definition safe (S A: Type)(o: outcome S A): Prop :=
  sat o (fun _ _ ⇒ True).
```

3.5 Mutators

```
Definition mutator S0 S1 A1 := S0 → outcome S1 A1.
Definition yield S A (a: A): mutator S S A := fun st ⇒ single st a.
Definition noop{S}: mutator S S unit :=
  fun s ⇒ single s tt.
Definition message_mut{S}(m: string): mutator S S unit :=
  fun s ⇒ message m (single s tt).
Definition fail{S0}{S1}{A1}: mutator S0 S1 A1 := fun _ ⇒ ofail.
Definition block{S0}{S1}{A1}: mutator S0 S1 A1 := fun _ ⇒ oblock.
Definition fork{S0}{S1}{A1}(op1 op2: mutator S0 S1 A1): mutator S0 S1 A1 :=
  fun s0 ⇒ ofork (op1 s0) (op2 s0).
Definition pick_demonic {S} T: mutator S S T :=
  fun st ⇒ demonicT (fun x ⇒ single st x).
```

3.6 Sequential composition of mutators

```
Fixpoint bindf S1 A1 (o: outcome S1 A1) S2 A2 (f: A1 → mutator S1 S2 A2): outcome
S2 A2 :=
```

```
  match o with
  | single s a ⇒ f a s
  | demonic os ⇒ demonic { bindf o f | o in os }
  | angelic os ⇒ angelic { bindf o f | o in os }
  | message msg o ⇒ message msg (bindf o f)
  end.

Definition bind S1 A1 (o: outcome S1 A1) S2 A2 (C: S1 → outcome S2 A2): outcome
S2 A2 :=
  bindf o (fun _ ⇒ C).

Definition seqf S0 S1 A1 S2 A2 (C: mutator S0 S1 A1) (f: A1 → mutator S1 S2 A2):
mutator S0 S2 A2 :=
  fun st ⇒ bindf (C st) f.
Definition seq S0 S1 A1 S2 A2 (C: mutator S0 S1 A1) (C': mutator S1 S2 A2): mutator
S0 S2 A2 :=
  seqf C (fun _ ⇒ C').

Notation "x <- op1 ; op2" := (seqf op1 (fun x ⇒ op2)) (at level 10, op1 at level 28,
op2 at level 30).
Notation "op1 ; op2" := (seq op1 op2) (at level 30, right associativity).

Definition seqcomma S0 S1 A1 S2 A2 (C: mutator S0 S1 A1) (C': mutator S1 S2 A2):
mutator S0 S2 A1 :=
  x ← C; C'; yield x.

Infix ";," := seqcomma (at level 25).

Fixpoint iter{S A}(C: mutator S S A)(n: nat): mutator S S (list A) :=
  match n with
    0 ⇒ yield nil
  | S n ⇒ v ← C; vs ← iter C n; yield (v::vs)
  end.
```

3.7 Coverage of outcomes

```
Definition ocovers{S A}(o1 o2: outcome S A): Prop :=
  ∀ Q, sat o1 Q → sat o2 Q.

Definition covers{S0 S1 A1}(C1 C2: S0 → outcome S1 A1): Prop :=
  ∀ st, ocovers (C1 st) (C2 st).

Infix "==>" := covers (at level 55, right associativity).

Definition oequiv{S A}(o1 o2: outcome S A): Prop :=
  ocovers o1 o2 ∧ ocovers o2 o1.
```

3.8 Some properties

```
Lemma outcome_ind'{S A}
  (P: outcome S A → Prop)
  (Hsingle: ∀ s a, P (single s a))
  (Hdemonic: ∀ os, forall_ os P → P (demonic os))
  (Hangelic: ∀ os, forall_ os P → P (angelic os))
  (Hmessage: ∀ msg o, P o → P (message msg o)):
  ∀ o, P o.

Lemma covers_refl S0 S1 A1 (C: mutator S0 S1 A1):
  C ==> C.

Lemma covers_trans
  S0 S1 A1
  (C1 C2 C3: mutator S0 S1 A1):
  C1 ==> C2 →
  C2 ==> C3 →
  C1 ==> C3.

Lemma bindf_Q S1 A1 (o: outcome S1 A1) S2 A2 (f: A1 → S1 → outcome S2 A2) Q:
  sat (bindf o f) Q ↔ sat o (fun s a ⇒ sat (f a s) Q).

Lemma sat_mono S A (o: outcome S A) (Q1 Q2: S → A → Prop):
  (∀ s a, Q1 s a → Q2 s a) →
  sat o Q1 → sat o Q2.

Add Parametric Relation S0 S1 A1:
  (mutator S0 S1 A1) covers
  reflexivity proved by (@covers_refl S0 S1 A1)
  transitivity proved by (@covers_trans S0 S1 A1)
  as covers_relation.

Lemma eq_covers {S0 S1 A1} {C C': S0 → outcome S1 A1}: C = C' → C ==> C'.

Lemma seqf_mono S0 S1 A1 S2 A2
  (C C': mutator S0 S1 A1) (f f': A1 → mutator S1 S2 A2):
  C ==> C' →
  (∀ x, f x ==> f' x) →
  seqf C f ==> seqf C' f'.

Lemma seqf_mono_l S0 S1 A1 S2 A2
  (C C': mutator S0 S1 A1) (f: A1 → mutator S1 S2 A2):
  C ==> C' →
  seqf C f ==> seqf C' f.

Lemma seqf_mono_r S0 S1 A1 S2 A2
  (C: mutator S0 S1 A1) (f f': A1 → mutator S1 S2 A2):
  (∀ x, f x ==> f' x) →
```

seqf *C f* ==> seqf *C f'*.

Lemma seq_mono *S0 S1 A1 S2 A2*
(*C1 C1'*: mutator *S0 S1 A1*) (*C2 C2'*: mutator *S1 S2 A2*):
C1 ==> *C1'* →
C2 ==> *C2'* →
C1 ; *C2* ==> *C1'* ; *C2'*.

Lemma seq_mono_l *S0 S1 A1 S2 A2*
(*C1 C1'*: mutator *S0 S1 A1*) (*C2*: mutator *S1 S2 A2*):
C1 ==> *C1'* →
C1 ; *C2* ==> *C1'* ; *C2*.

Lemma seq_mono_r *S0 S1 A1 S2 A2*
(*C1*: mutator *S0 S1 A1*) (*C2 C2'*: mutator *S1 S2 A2*):
C2 ==> *C2'* →
C1 ; *C2* ==> *C1* ; *C2'*.

Lemma seqcomma_mono *S0 S1 A1 S2 A2*
(*C1 C1'*: mutator *S0 S1 A1*) (*C2 C2'*: mutator *S1 S2 A2*):
C1 ==> *C1'* →
C2 ==> *C2'* →
C1 ;, *C2* ==> *C1'* ;, *C2'*.

Lemma bindf_assoc *S1 A1 S2 A2 S3 A3*
(*o*: **outcome** *S1 A1*)
(*f*: *A1* → *S1* → **outcome** *S2 A2*)
(*g*: *A2* → *S2* → **outcome** *S3 A3*):
bindf (bindf *o f*) *g* = bindf *o* (`fun` *x s* ⇒ bindf (*f x s*) *g*).

Lemma seqf_seqf_assoc *S0 S1 A1 S2 A2 S3 A3*
(*C*: *S0* → **outcome** *S1 A1*)
(*f*: *A1* → *S1* → **outcome** *S2 A2*)
(*g*: *A2* → *S2* → **outcome** *S3 A3*):
seqf (seqf *C f*) *g* = seqf *C* (`fun` *x* ⇒ seqf (*f x*) *g*).

Lemma seq_seqf_assoc *S0 S1 A1 S2 A2 S3 A3*
(*C*: *S0* → **outcome** *S1 A1*)
(*f*: *A1* → *S1* → **outcome** *S2 A2*)
(*C'*: *S2* → **outcome** *S3 A3*):
seq (seqf *C f*) *C'* = seqf *C* (`fun` *x* ⇒ seq (*f x*) *C'*).

Lemma seq_assoc *S0 S1 A1 S2 A2 S3 A3*
(*C1*: *S0* → **outcome** *S1 A1*)
(*C2*: *S1* → **outcome** *S2 A2*)
(*C3*: *S2* → **outcome** *S3 A3*):
(*C1* ; *C2*) ; *C3* = *C1* ; *C2* ; *C3*.

Lemma seqf_seq_assoc *S0 S1 A1 S2 A2 S3 A3*

```
  (C: S0 → outcome S1 A1)
  (C': S1 → outcome S2 A2)
  (f: A2 → S2 → outcome S3 A3):
  seqf (seq C C') f = seq C (seqf C' f).

Definition op_equiv{S0 S1 A1}(C1 C2: S0 → outcome S1 A1): Prop :=
  C1 ==> C2 ∧ C2 ==> C1.

Infix "<==>" := op_equiv (at level 55).

Import Setoid.
Import Morphisms.

Add Parametric Morphism U S A T B: (@seq U S A T B)
  with signature covers ++> covers ++> covers as seq_morphism.
Qed.

Add Parametric Morphism U S A T B: (@seqf U S A T B)
  with signature covers ++> pointwise_relation A covers ++> covers as seqf_morphism.
Qed.

Add Parametric Morphism U S A T B: (@seqcomma U S A T B)
  with signature covers ++> covers ++> covers as seqcomma_morphism.
Qed.

Instance reflexive_properproxy_pointwise_relation
  A B (R: relation B) (x: A → B) `(Reflexive B R):
  ProperProxy (pointwise_relation A R) x.
Qed.

Instance reflexive_properproxy_inverse_pointwise_relation
  A B (R: relation B) (x: A → B) `(Reflexive B R):
  ProperProxy (inverse (pointwise_relation A R)) x.
Qed.

Lemma pick_demonic_covers_fork U S A (C1 C2: U → outcome S A):
  b ← pick_demonic bool; (if b then C1 else C2) ==> fork C1 C2.

Lemma fork_covers_pick_demonic U S A (C1 C2: U → outcome S A):
  fork C1 C2 ==> b ← pick_demonic bool; (if b then C1 else C2).

Lemma pick_demonic_elim S A a: pick_demonic A ==> yield (S:=S) a.

Lemma seqf_yield_elim S1 A1 S2 A2 a (f: A1 → S1 → outcome S2 A2):
  seqf (yield a) f = f a.

Lemma seq_pick_demonic_l S0 S1 S2 (C: mutator S0 S1 unit) A B (f: A → mutator S1 S2
B):
  seq C (seqf (pick_demonic A) f) ==> seqf (pick_demonic A) (fun x ⇒ seq C (f x)).

Lemma seqcomma_unit
  S1 S2 S3
```

(C: S1 → **outcome** S2 **unit**)
(C': S2 → **outcome** S3 **unit**):
C ;, C' ==> C ; C'.

Lemma seqcomma_unit'
S1 S2 S3
(C: S1 → **outcome** S2 **unit**)
(C': S2 → **outcome** S3 **unit**):
C ; C' ==> C ;, C'.

Definition single_mut {S0 S1 A1} (C: mutator S0 S1 A1) :=
∀ st, ∃ st' a, oequiv (C st) (single st' a).

Definition demonic_mut {U S A} (C: U → **outcome** S A) :=
∀ st,
∃ T (fst: T → S) (fa: T → A),
∀ Q, sat (C st) Q ↔ ∀ (t: T), Q (fst t) (fa t).

Lemma oequiv_refl S A (o: **outcome** S A): oequiv o o.

Lemma single_mut_demonic_mut S0 S1 A1 (C: mutator S0 S1 A1): single_mut C → demonic_mut C.

Lemma seq_pick_demonic E U S A (C: U → **outcome** S A) T B (f: E → S → **outcome** T B):
demonic_mut C →
x ← pick_demonic E; C ; f x ==> C ; seqf (pick_demonic E) f.

Lemma covers_pick_demonic:
∀ U S A (C: U → **outcome** S A) T (f: T → U → **outcome** S A),
(∀ x, C ==> f x) →
C ==> seqf (pick_demonic T) f.

Lemma pick_demonic_covers U S A T (f: T → U → **outcome** S A) (C: U → **outcome** S A) t:
f t ==> C →
seqf (pick_demonic T) f ==> C.

Lemma covers_yield_yield S A (x: A) B (y: B):
(yield y: mutator S S B) ==> yield x ; yield y.

Lemma op_equiv_symm S0 S1 A1 (C C': mutator S0 S1 A1): C <==> C' → C' <==> C.

Lemma yield_seq A U S B (C: U → **outcome** S B) (x: A): C ==> yield x ; C.

Lemma yield_seqf A U S B (f: A → U → **outcome** S B) (x: A): f x ==> y ← yield x ; f y.

Lemma prepend_noop S0 S1 A1 (C: mutator S0 S1 A1):
C ==> noop; C.

Definition pick_angelic {S} T: mutator S S T :=
fun st ⇒ angelicT (fun x ⇒ single st x).

Lemma pick_angelic_covers T $S0$ S A (f: $T \to S0 \to$ **outcome** S A) (C: $S0 \to$ **outcome** S A):
  ($\forall$ x, f x ==> C) $\to$
  seqf (pick_angelic T) f ==> C.

Definition angelic_mut {$S0$ $S1$ $A1$} (C: mutator $S0$ $S1$ $A1$) :=
  $\exists$ X,
  $\forall$ st Q, sat (C st) Q $\leftrightarrow$ ($\exists$ st' a, X st st' a $\wedge$ Q st' a).

Lemma single_mut_angelic_mut $S0$ $S1$ $A1$ (C: mutator $S0$ $S1$ $A1$): single_mut C $\to$ angelic_mut C.

Lemma angelic_mut_fail {$S0$ $S1$ $A1$ $S2$ $A2$} (C: mutator $S0$ $S1$ $A1$): angelic_mut C $\to$ seq C fail ==> (fail: mutator $S0$ $S2$ $A2$).

Lemma angelic_mut_yield S A (a: A): angelic_mut (yield (S:=S) a).

Lemma angelic_mut_seqf {$S0$ $S1$ $A1$ $S2$ $A2$} (C: mutator $S0$ $S1$ $A1$) (f: $A1 \to$ mutator $S1$ $S2$ $A2$):
  angelic_mut C $\to$ ($\forall$ x, angelic_mut (f x)) $\to$ angelic_mut (seqf C f).

Lemma safe_pick_demonic:
  $\forall$ T $S0$ $S1$ $A1$ x (f: $T \to$ mutator $S0$ $S1$ $A1$) st,
  safe (seqf (pick_demonic T) f st) $\to$
  safe (f x st).

Lemma sat_iter_pick_demonic n $S0$ $S1$ $A1$ (f: _ $\to$ mutator $S0$ $S1$ $A1$) vs st Q:
  sat (seqf (iter (pick_demonic **Z**) n) f st) Q $\to$
  length vs = n $\to$
  sat (f vs st) Q.

Lemma safe_iter_pick_demonic n $S0$ $S1$ $A1$ (f: _ $\to$ mutator $S0$ $S1$ $A1$) vs st:
  safe (seqf (iter (pick_demonic **Z**) n) f st) $\to$
  length vs = n $\to$
  safe (f vs st).

Lemma not_covers_fail $S0$ $S1$ A (C: mutator $S0$ $S1$ A):
  $\neg$ C ==> fail $\to$ $\exists$ st, safe (C st).

Lemma covers_block $S0$ $S1$ A (C: mutator $S0$ $S1$ A): C ==> block.

Lemma covers_fork $S0$ $S1$ A ($C1$ $C2$ $C3$: mutator $S0$ $S1$ A):
  $C1$ ==> $C2$ $\to$ $C1$ ==> $C3$ $\to$ $C1$ ==> fork $C2$ $C3$.

Lemma seq_noop_covers {$S0$ $S1$} (C: mutator $S0$ $S1$ **unit**): C; noop ==> C.

Lemma noop_seq_covers {$S0$ $S1$ A} (C: mutator $S0$ $S1$ A): noop; C ==> C.

Lemma noop_intro $S0$ $S1$ $A1$ (C: mutator $S0$ $S1$ $A1$):
  C ==> noop; C.

Lemma noop_elim $S0$ $S1$ $A1$ (C: mutator $S0$ $S1$ $A1$):
  noop; C ==> C.

Lemma seq_noop_intro *S0 S1 A1* (*C1*: mutator *S0 S1 A1*) *S2 A2* (*C2*: mutator *S1 S2 A2*):
C1 ; *C2* ==> (*C1* ; noop) ; *C2*.

Lemma seq_noop_elim *S0 S1 A1* (*C1*: mutator *S0 S1 A1*) *S2 A2* (*C2*: mutator *S1 S2 A2*):
(*C1* ; noop) ; *C2* ==> *C1* ; *C2*.

Definition string := **String.string**.
Definition string_dec := String.string_dec.

Chapter 4

Library ConcreteExecution

```
Require Export Programs.
Require Export Outcomes.
```

4.1 Concrete states

```
Inductive cchunk_id := points_to_id(l: Z) | malloc_block_id(l: Z).
Definition cchunk_id_dec (i1 i2: cchunk_id): {i1 = i2} + {i1 ≠ i2}.
Defined.
Definition cheap := cchunk_id → option Z.
Definition cstate := (store × cheap)%type.
Definition cheap_update (h: cheap) l v l' := if cchunk_id_dec l l' then v else h l'.
Section RoutineDefs.
Variable routine_defs: routine_table.
```

4.2 Concrete outcomes

```
Definition coutcome A := outcome cstate A.
Definition cmutatora A := cstate → coutcome A.
Definition cmutator := cmutatora unit.
Definition pick{S}: mutator S S Z :=
  fun st ⇒ demonic (set_img Z_set (fun z ⇒ single st z)).
```

4.3 Concrete basic mutators

```
Definition assumeb(b: bool): cmutator :=
```

```
  fun st ⇒ if b then single st tt else oblock.
Definition update_cstore x v: cmutator :=
  fun st ⇒ let (s, h) := st in
  single (store_update s x v, h) tt.
Definition cwith_store s {A} (C: cmutatora A): cmutatora A :=
  fun st ⇒ let (s0, h) := st in
  bindf (C (s, h)) (fun a st' ⇒ let (_, h') := st' in single (s0, h') a).
Definition ceval_mut(e: expr): cmutatora Z :=
  fun st ⇒ let (s, h) := st in
  single st (eval s e).
Definition cevals_mut(es: list expr) :=
  fun st: cstate ⇒ let (s, h) := st in
  single st (map (eval s) es).
Definition cassume_bexpr(b: bexpr): cmutator :=
  fun st ⇒ let (s, h) := st in
  if beval s b then single st tt else oblock.
Definition ccons_chunk i: cmutatora Z :=
  fun st ⇒ let (s, h) := st in
  match h i with
    None ⇒ ofail
  | Some v ⇒ single (s, cheap_update h i None) v
  end.
Definition ccons_pointsto l: cmutatora Z := ccons_chunk (points_to_id l).
Definition ccons_malloc_block l: cmutatora Z := ccons_chunk (malloc_block_id l).
Fixpoint ccons_pointstos l n :=
  match n with
    O ⇒ noop
  | S n ⇒
    ccons_pointsto l;
    ccons_pointstos (l + 1) n
  end.
Definition cprod_chunk i v: cmutator :=
  fun st ⇒ let (s, h) := st in
  match h i with
    None ⇒ single (s, cheap_update h i (Some v)) tt
  | Some _ ⇒ oblock
  end.
Definition cprod_pointsto l v: cmutator := cprod_chunk (points_to_id l) v.
Fixpoint cprod_pointstos l n :=
```

```
  match n with
    O ⇒ noop
  | S n ⇒
    v ← pick;
    cprod_pointsto l v;
    cprod_pointstos (l + 1) n
  end.

Fixpoint fpow(n: nat){A}(f: A → A)(x: A): A :=
  match n with
    O ⇒ x
  | S n ⇒ f (fpow n f x)
  end.

Definition iterate_n(n: nat){S}(C: mutator S S unit): mutator S S unit :=
  fpow n (fun C' ⇒ fork (C; C') noop) block.

Definition iterate{S}(C: mutator S S unit): mutator S S unit :=
  n ← pick;
  iterate_n (Z.to_nat n) C.
```

4.4 Concrete execution

```
Fixpoint c_exec_n(n: nat)(c: cmd): cmutator :=
  match n with
    O ⇒ block
  | S n ⇒
    match c with
    | Assign x e ⇒
      v ← ceval_mut e;
      update_cstore x v
    | Malloc x n ⇒
      l ← pick;
      update_cstore x l;
      assumeb (Z_ltb 0 l);
      cprod_chunk (malloc_block_id l) (Z.of_nat n);
      cprod_pointstos l n
    | Free e ⇒
      l ← ceval_mut e;
      n ← ccons_malloc_block l;
      ccons_pointstos l (Z.to_nat n)
    | Read x e ⇒
      l ← ceval_mut e;
      v ← ccons_pointsto l;
```

```
        cprod_pointsto l v;
        update_cstore x v
      | Write e1 e2 ⇒
        l ← ceval_mut e1;
        v ← ceval_mut e2;
        _ ← ccons_pointsto l;
        cprod_pointsto l v
      | Call x r es ⇒
        match routine_defs r with
          None ⇒ fail
        | Some (RoutineDef xs c') ⇒
          vs ← cevals_mut es;
          v ← cwith_store (store_updates store0 xs vs) (c_exec_n n c'; ceval_mut result);
          update_cstore x v
        end
      | IfCmd b c1 c2 ⇒
        fork (
          cassume_bexpr b;
          c_exec_n n c1
        ) (
          cassume_bexpr (b_not b);
          c_exec_n n c2
        )
      | While b a c ⇒
        iterate (
          cassume_bexpr b;
          c_exec_n n c
        );
        cassume_bexpr (b_not b)
      | Seq c1 c2 ⇒
        c_exec_n n c1;
        c_exec_n n c2
      | Open p es ⇒ noop
      | Close p es ⇒ noop
      | Skip ⇒ noop
      | Message m ⇒ message_mut m
      end
    end.

Definition c_exec(c: cmd): cmutator :=
  n ← pick;
  c_exec_n (Z.to_nat n) c.

Definition cheap0: cheap := fun _ ⇒ None.
```

```
Definition cstate0 := (store0, cheap0).

Definition cvalid_program(c: cmd): Prop :=
  safe (cstate0 |> c_exec c).
```

4.5 Exploring concrete execution outcomes

```
Fixpoint is_ofail{A}(o: coutcome A): Prop :=
  match o with
    angelic (set_ n_Empty_set _) ⇒ True
  | _ ⇒ False
  end.

Fixpoint subformula{A}(f: coutcome A)(ps: list Z) {struct ps}: coutcome A :=
  match ps with
    nil ⇒ f
  | p::ps ⇒
    match f with
      demonic (set_ n_Z F) ⇒ subformula (F p) ps
    | demonic (set_ n_bool F) ⇒
      if Z_eq_dec p 0 then
        subformula (F true) ps
      else
        subformula (F false) ps
    | _ ⇒ f
    end
  end.

Lemma is_ofail_subformula_not_safe A ps:
  ∀ (o: coutcome A),
  is_ofail (subformula o ps) → ¬ safe o.

End RoutineDefs.

Open Local Scope string_scope.

Inductive outcome_kind :=
  o_single | o_block | o_fork | o_demonic | o_fail | o_fork_angelic | o_angelic | o_message(m:
string).

Fixpoint get_outcome_kind{A}(o: coutcome A): outcome_kind :=
  match o with
    single _ _ ⇒ o_single
  | demonic (set_ n_Empty_set _) ⇒ o_block
  | demonic (set_ n_bool _) ⇒ o_fork
  | demonic _ ⇒ o_demonic
  | angelic (set_ n_Empty_set _) ⇒ o_fail
```

```
  | angelic (set_ n_bool _) ⇒ o_fork_angelic
  | angelic _ ⇒ o_angelic
  | message m _ ⇒ o_message m
  end.

Fixpoint get_outcome_kinds{A}(f: coutcome A)(ps: list Z): list (Z × outcome_kind) :=
  match ps with
    nil ⇒ nil
  | p::ps ⇒
    (p, get_outcome_kind f)::
    match f with
      demonic (set_ n_Z F) ⇒ get_outcome_kinds (F p) ps
    | demonic (set_ n_bool F) ⇒
      if Z_eq_dec p 0 then
        get_outcome_kinds (F true) ps
      else
        get_outcome_kinds (F false) ps
    | message m o ⇒ get_outcome_kinds o ps
    | _ ⇒ nil
    end
  end.

Definition is_single{A}(o: coutcome A): Prop :=
  match o with
    single _ _ ⇒ True
  | _ ⇒ False
  end.
```

4.6 Example concrete executions

```
Local Notation x := "x".
Local Notation y := "y".
Local Notation z := "z".

Definition rt0: routine_table := fun _ ⇒ None.

Goal is_ofail (c_exec_n rt0 1 (x ':=' [0]) cstate0).

Definition exec0 c := c_exec rt0 c cstate0.

Definition assert b := (If b Then Skip Else call "fail"())%cmd.

Goal is_single
  (subformula
    (exec0 (
       x ':=' malloc(2); [x] ':=' 0; [x + 1] ':=' 1;
       y ':=' [x + 1];
```

```
      assert (y == 1)
    ))
    [10, 17, 42, 19, 0]).
Goal is_ofail
  (subformula
    (exec0 (
      x ':=' malloc(2); [x] ':=' 0; [x + 1] ':=' 1;
      y ':=' [x + 1];
      assert (y == 0)
    ))
    [10, 17, 42, 19, 1]).
Local Notation a := "a".
Local Notation b := "b".
Local Notation tail := "tail".
```

Constructs a linked list whose values range from a, inclusive, to b, exclusive. `Definition make_range :=`

```
  RoutineDef [a, b] (
    If a == b Then
      result ':=' 0
    Else (
      tail ':=' call "make_range"(a + 1, b);
      result ':=' malloc(2); [result] ':=' a; [result + 1] ':=' tail
    )
  ).
```

Prepends the reverse of the linked list at a to the linked list at b in place and returns a pointer to the resulting linked list. `Definition reverse :=`

```
  RoutineDef [a, b] (
    If a == 0 Then
      result ':=' b
    Else (
      tail ':=' [a + 1];
      [a + 1] ':=' b;
      result ':=' call "reverse"(tail, a)
    )
  ).
Definition dispose :=
  RoutineDef [a] (
    If a == 0 Then
      Skip
    Else (
      tail ':=' [a + 1];
```

```
      free(a);
      call "dispose"(tail)
    )
  ).
Definition reverse_loop :=
  RoutineDef [a] (
    while ¬ (a == 0) invariant b_true do (
      tail ':=' [a + 1];
      [a + 1] ':=' b;
      b ':=' a;
      a ':=' tail
    );
    result ':=' b
  ).
Definition dispose_loop :=
  RoutineDef [a] (
    while ¬ (a == 0) invariant b_true do (
      tail ':=' [a + 1];
      free(a);
      a ':=' tail
    )
  ).
Definition my_rt :=
  fun r ⇒
    if string_dec r "make_range" then Some make_range else
    if string_dec r "reverse" then Some reverse else
    if string_dec r "reverse_loop" then Some reverse_loop else
    if string_dec r "dispose" then Some dispose else
    if string_dec r "dispose_loop" then Some dispose_loop else
    None.
Definition reverse_test :=
  (
    a ':=' call "make_range"(0, 5);
    b ':=' call "reverse"(a, 0);
    a ':=' call "reverse"(b, 0);
    call "dispose"(a)
  )%cmd.
Goal
  is_single
    (subformula
       (c_exec my_rt reverse_test cstate0)
```

```
      [200, 1, 1, 1, 1, 1, 0, 410, 9, 9, 310, 9, 9, 210, 9, 9, 110, 9, 9, 10, 9, 9, 1,
1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0]).

Definition reverse_test_broken :=
  (
    a ':=' call "make_range"(0, 5);
    b ':=' call "reverse"(a, 0);
    call "reverse"(b + 10, 0)
  )%cmd.

Goal
  is_ofail
    (subformula
      (c_exec my_rt reverse_test_broken cstate0)
      [200, 1, 1, 1, 1, 1, 0, 410, 9, 9, 310, 9, 9, 210, 9, 9, 110, 9, 9, 10, 9, 9, 1,
1, 1, 1, 1, 0, 1]).

Definition reverse_loop_test :=
  (
    a ':=' call "make_range"(0, 5);
    b ':=' call "reverse_loop"(a, 0);
    a ':=' call "reverse_loop"(b, 0);
    call "dispose_loop"(a)
  )%cmd.

Goal
  is_single
    (subformula
      (c_exec my_rt reverse_loop_test cstate0)
      [200, 1, 1, 1, 1, 1, 0, 410, 9, 9, 310, 9, 9, 210, 9, 9, 110, 9, 9, 10, 9, 9,
50, 0, 0, 0, 0, 0, 1, 50, 0, 0, 0, 0, 0, 1, 50, 0, 0, 0, 0, 0, 1]).

Definition reverse_loop_test_broken :=
  (
    a ':=' call "make_range"(0, 5);
    b ':=' call "reverse_loop"(a, 0);
    call "reverse_loop"(b + 10, 0)
  )%cmd.

Goal
  is_ofail
    (subformula
      (c_exec my_rt reverse_loop_test_broken cstate0)
      [200, 1, 1, 1, 1, 1, 0, 410, 9, 9, 310, 9, 9, 210, 9, 9, 110, 9, 9, 10, 9, 9,
50, 0, 0, 0, 0, 0, 1, 50, 0]).
```

Chapter 5

Library SemiconcreteStates

```
Require Export Programs.
Require Export FunctionalExtensionality.
Local Open Scope nat_scope.
```

5.1 Semiconcrete states

```
Inductive chunk := Chunk(p: pred)(vs: list value).
Definition pointsto l v := Chunk pointsto_pred [l, v].
Definition malloc_block l v := Chunk malloc_block_pred [l, v].
Definition heap := chunk → nat.
Definition state := (store × heap)%type.
Definition empty_heap: heap := fun c ⇒ O.
Definition state0 := (store0, empty_heap).
```

5.2 Working with semiconcrete heaps

```
Definition chunk_dec (c1 c2: chunk): {c1 = c2} + {c1 ≠ c2}.
Defined.
Definition singleton_heap(c: chunk): heap :=
  fun c' ⇒ if chunk_dec c c' then 1 else 0.
Definition heap_add(h1 h2: heap): heap :=
  fun c ⇒ h1 c + h2 c.
```

5.3 Properties of the semiconcrete heap constructors

```
Lemma singleton_heap_same: ∀ c, singleton_heap c c = 1.
```

Lemma singleton_heap_diff: $\forall$ c c', $c \neq c'$ $\rightarrow$ singleton_heap c c' = 0.

Lemma heap_add_assoc: $\forall$ $h1$ $h2$ $h3$, heap_add (heap_add $h1$ $h2$) $h3$ = heap_add $h1$ (heap_add $h2$ $h3$).

Lemma heap_add_commut: $\forall$ $h1$ $h2$, heap_add $h1$ $h2$ = heap_add $h2$ $h1$.

Lemma heap_add_empty_l: $\forall$ h, heap_add empty_heap h = h.

Lemma heap_add_empty_r: $\forall$ h, heap_add h empty_heap = h.

Lemma heap_add_singleton0:
  $\forall$ $h1$ $h2$ c h n,
  heap_add $h1$ $h2$ = heap_add (singleton_heap c) h $\rightarrow$
  $h2$ c = S n $\rightarrow$
  ($\exists$ $h2'$, $h2$ = heap_add (singleton_heap c) $h2'$ $\wedge$ h = heap_add $h1$ $h2'$).

Lemma heap_add_singleton:
  $\forall$ $h1$ $h2$ c h,
  heap_add $h1$ $h2$ = heap_add (singleton_heap c) h $\rightarrow$
  ($\exists$ $h1'$, $h1$ = heap_add (singleton_heap c) $h1'$ $\wedge$ h = heap_add $h1'$ $h2$) $\vee$
  ($\exists$ $h2'$, $h2$ = heap_add (singleton_heap c) $h2'$ $\wedge$ h = heap_add $h1$ $h2'$).

Lemma singleton_heap_add_singleton:
  $\forall$ $c1$ $c2$ h,
  singleton_heap $c1$ = heap_add (singleton_heap $c2$) h $\rightarrow$
  $c1$ = $c2$ $\wedge$ h = empty_heap.

Lemma heap_add_eq_empty:
  $\forall$ $h1$ $h2$,
  heap_add $h1$ $h2$ = empty_heap $\leftrightarrow$
  $h1$ = empty_heap $\wedge$ $h2$ = empty_heap.

Lemma singleton_heap_add:
  $\forall$ c $h1$ $h2$,
  singleton_heap c = heap_add $h1$ $h2$ $\rightarrow$
  ($h1$ = singleton_heap c $\wedge$ $h2$ = empty_heap) $\vee$
  ($h2$ = singleton_heap c $\wedge$ $h1$ = empty_heap).

Lemma heap_add_heap_add:
  $\forall$ $h1$ $h2$ ha hb,
  heap_add $h1$ $h2$ = heap_add ha hb $\rightarrow$
  $\exists$ $h1a$, $\exists$ $h1b$, $\exists$ $h2a$, $\exists$ $h2b$,
  $h1$ = heap_add $h1a$ $h1b$ $\wedge$
  $h2$ = heap_add $h2a$ $h2b$ $\wedge$
  ha = heap_add $h1a$ $h2a$ $\wedge$
  hb = heap_add $h1b$ $h2b$.

Lemma singleton_neq_empty:
  $\forall$ c,
  singleton_heap c $\neq$ empty_heap.

Chapter 6

Library SemiconcreteExecution

```
Require Export Bool.
Require Export SetoidDec.
Require Export SemiconcreteStates.
Require Export List.
Require Export Outcomes.
```

6.1 Semiconcrete outcomes

```
Definition scoutcome A := outcome state A.
Definition scmutatora A := state → scoutcome A.
Definition scmutator := scmutatora unit.
```

6.2 Semiconcrete basic mutators

```
Definition assume_prop(P: Prop): scmutator :=
  fun st ⇒
  demonicT (fun H: P ⇒ single st tt).

Definition update_store(x: var)(v: Z): scmutator :=
  fun st ⇒ let (s, h) := st in
  single (store_update s x v , h) tt.

Definition update_store_n(xs: list var)(vs: list Z): scmutator :=
  fun st ⇒ let (s, h) := st in
  single (store_updates s xs vs , h) tt.

Definition havoc1(x: var): scmutator :=
  v ← pick_demonic Z;
  update_store x v.

Fixpoint havoc(xs: list var): scmutator :=
```

```
  match xs with
    [] ⇒ noop
  | x::xs ⇒ havoc1 x; havoc xs
  end.
Definition with_store{A}(s: store)(op: scmutatora A): scmutatora A :=
  fun st ⇒ let (s0, h) := st in
  bindf (op (s, h)) (fun a st ⇒ let (_, h) := st in single (s0, h) a).
Definition with_store' s (C: scmutator): scmutatora store :=
  fun st ⇒ let (s0, h) := st in
  bind (C (s, h)) (fun st' ⇒ let (s', h') := st' in single (s0, h') s').
Definition with_local_store{A}(op: scmutatora A): scmutatora A :=
  fun st ⇒ let (s, h) := st in
  bindf (op (s, h)) (fun a st ⇒ let (_, h) := st in single (s, h) a).
Definition eval_mut(e: expr): scmutatora Z :=
  fun st ⇒ let (s, h) := st in
  single st (eval s e).
Definition evals_mut(es: list expr) :=
  fun st: state ⇒ let (s, h) := st in
  single st (map (eval s) es).
Definition beval_mut(b: bexpr): scmutatora bool :=
  fun st ⇒ let (s, h) := st in
  single st (beval s b).
Definition assume_bexpr(b: bexpr): scmutator :=
  fun st ⇒ let (s, h) := st in
  if beval s b then single st tt else oblock.
Definition assert_bexpr(b: bexpr): scmutator :=
  fun st ⇒ let (s, h) := st in
  if beval s b then
    single st tt
  else
    ofail.
Definition clear_heap: scmutator :=
  fun st ⇒ let (s, h) := st in
  single (s, empty_heap) tt.
Definition leakcheck: scmutator :=
  fun st ⇒ let (s, h) := st in
  angelicT (fun H: h = empty_heap ⇒
    oblock
  ).
Definition cons_chunk p ls n: scmutatora (list Z) :=
```

```
  fun st ⇒ let (s, h) := st in
  angelicT (fun vs ⇒
  angelicT (fun Hlength: length vs = n ⇒
  angelicT (fun h' ⇒
  angelicT (fun _: h = heap_add (singleton_heap (Chunk p (ls ++ vs))) h' ⇒
    single (s, h') vs
  )))).
Definition cons_chunk_1_1 p l: scmutatora Z := vs ← cons_chunk p [l] 1; yield (hd 0 vs).
Definition cons_pointsto := cons_chunk_1_1 pointsto_pred.
Definition cons_malloc_block := cons_chunk_1_1 malloc_block_pred.

Fixpoint cons_pointstos l n :=
  match n with
    O ⇒ noop
  | S n ⇒
    cons_pointsto l;
    cons_pointstos (l + 1) n
  end.

Definition prod_chunk(c: chunk): scmutator :=
  fun st ⇒ let (s, h) := st in
  single (s, heap_add (singleton_heap c) h) tt.

Fixpoint prod_pointstos(l: Z)(n: nat): scmutator :=
  match n with
    O ⇒ noop
  | S n ⇒
    v ← pick_demonic Z;
    prod_chunk (pointsto l v); prod_pointstos (l + 1) n
  end.
```

6.3 Semiconcrete execution

```
Fixpoint consume(a: asn): scmutator :=
  match a with
    BAsn b ⇒ assert_bexpr b
  | PAsn p es xs ⇒
    ls ← evals_mut es;
    vs ← cons_chunk p ls (length xs);
    update_store_n xs vs
  | IfAsn b a1 a2 ⇒
    fork (
      assume_bexpr b;
      consume a1
```

```
    ) (
      assume_bexpr (b_not b);
      consume a2
    )
  | SepAsn a1 a2 ⇒
    consume a1;
    consume a2
  end.

Fixpoint produce(a: asn): scmutator :=
  match a with
    BAsn b ⇒ assume_bexpr b
  | PAsn p es xs ⇒
    ls ← evals_mut es;
    vs ← iter (pick_demonic Z) (length xs);
    update_store_n xs vs;
    prod_chunk (Chunk p (ls ++ vs))
  | IfAsn b a1 a2 ⇒
    fork (
      assume_bexpr b;
      produce a1
    ) (
      assume_bexpr (b_not b);
      produce a2
    )
  | SepAsn a1 a2 ⇒
    produce a1;
    produce a2
  end.

Section SemiconcreteExecution.

Variable routine_specs: spec_table.
Variable pred_defs: pred_table.

Fixpoint sc_exec(c: cmd): scmutator :=
  match c with
  | Assign x e ⇒
    v ← eval_mut e;
    update_store x v
  | Malloc x n ⇒
    l ← pick_demonic Z;
    update_store x l;
    assume_prop (0 < l)%Z;
    prod_chunk (malloc_block l (Z.of_nat n));
```

```
  prod_pointstos l n
| Free e ⇒
  l ← eval_mut e;
  n ← cons_malloc_block l;
  cons_pointstos l (Z.to_nat n)
| Read x e ⇒
  l ← eval_mut e;
  v ← cons_pointsto l;
  prod_chunk (pointsto l v);
  update_store x v
| Write e1 e2 ⇒
  l ← eval_mut e1;
  v ← eval_mut e2;
  _ ← cons_pointsto l;
  prod_chunk (pointsto l v)
| Call x r es ⇒
  match StringMap.find r routine_specs with
    None ⇒ fail
  | Some (RoutineSpec xs pre post) ⇒
    if length es == length xs then
      vs ← evals_mut es;
      s1 ← with_store' (store_updates store0 xs vs) (consume pre);
      v ← pick_demonic Z;
      with_store (store_update s1 result v) (produce post);
      update_store x v
    else
      fail
  end
| IfCmd b c1 c2 ⇒
  fork (
    assume_bexpr b;
    sc_exec c1
  ) (
    assume_bexpr (b_not b);
    sc_exec c2
  )
| While b a c ⇒
  with_local_store (consume a);
  havoc (targets c);
  fork (
    clear_heap; (
      with_local_store (produce a);
```

```
          assume_bexpr b;
          sc_exec c;
          with_local_store (consume a)
        );
        leakcheck
      ) (
        with_local_store (produce a);
        assume_bexpr (b_not b)
      )
  | Seq c1 c2 ⇒
      sc_exec c1;
      sc_exec c2
  | Open p es ⇒
      match pred_defs p with
        None ⇒ fail
      | Some (PredDef xs a) ⇒
        if le_dec (length es) (length xs) then
          ls ← evals_mut es;
          vs ← cons_chunk p ls (length xs - length es);
          with_store (store_updates store0 xs (ls ++ vs)) (
            produce a
          )
        else
          fail
      end
  | Close p es ⇒
      match pred_defs p with
        None ⇒ fail
      | Some (PredDef xs a) ⇒
        if length es == length xs then
          vs ← evals_mut es;
          with_store (store_updates store0 xs vs) (
            consume a
          );
          prod_chunk (Chunk p vs)
        else
          fail
      end
  | Skip ⇒ noop
  | Message m ⇒ noop
  end.
```

Section RoutineDefs.

```
Variable rdefs: routine_table.

Definition valid_routine r (rspec: routine_spec) :=
  match rdefs r with
    None ⇒ False
  | Some (RoutineDef xs c) ⇒
    let (xs', pre, post) := rspec in
      safe (
        state0 |>
        vs ← iter (pick_demonic Z) (length xs');
        with_store store0 (
          s1 ← with_store' (store_updates store0 xs' vs) (produce pre);
          v ← with_store (store_updates store0 xs vs) (sc_exec c; eval_mut result);
          with_store (store_update s1 result v) (consume post)
        );
        leakcheck
      )
  end.

Definition valid_routines :=
  Forall
    (fun el ⇒ valid_routine (fst el) (snd el))
    (StringMap.elements routine_specs).

Definition valid_command c :=
  safe (
    state0 |> sc_exec c
  ).

Definition valid_program c :=
  pred_defs pointsto_pred = None ∧
  pred_defs malloc_block_pred = None ∧
  valid_routines ∧
  valid_command c.

End RoutineDefs.

End SemiconcreteExecution.
```

Chapter 7

Library ConsumeProduce

```
Require Export SemiconcreteStates.
```

7.1 Consumption and production as relations between semiconcrete states

```
Inductive consume: state → asn → state → Prop :=
| consume_BAsn b s h:
  beval s b = true →
  consume (s, h) (BAsn b) (s, h)
| consume_PAsn p es xs s h vs:
  length vs = length xs →
  consume
    (s, heap_add (singleton_heap (Chunk p (map (eval s) es ++ vs))) h)
    (PAsn p es xs)
    (store_updates s xs vs, h)
| consume_IfAsn_true b a1 a2 s h st:
  beval s b = true →
  consume (s, h) a1 st →
  consume (s, h) (IfAsn b a1 a2) st
| consume_IfAsn_false b a1 a2 s h st:
  beval s b = false →
  consume (s, h) a2 st →
  consume (s, h) (IfAsn b a1 a2) st
| consume_SepAsn a1 a2 st st' st'':
  consume st a1 st' →
  consume st' a2 st'' →
  consume st (SepAsn a1 a2) st''.
```

```
Lemma consume_local:
```

```
  ∀ a s h s' h' h0,
  consume (s, h) a (s', h') →
  consume (s, heap_add h h0) a (s', heap_add h' h0).

Lemma consume_mono:
  ∀ a s h s' h',
  consume (s, h) a (s', h') →
  ∃ h'', h = heap_add h'' h' ∧ consume (s, h'') a (s', empty_heap).

Inductive produce: state → asn → state → Prop :=
| produce_BAsn b s h:
  beval s b = true →
  produce (s, h) (BAsn b) (s, h)
| produce_PAsn p es xs s h vs:
  length vs = length xs →
  produce
    (s, h)
    (PAsn p es xs)
    (store_updates s xs vs, heap_add (singleton_heap (Chunk p (map (eval s) es ++ vs))) h)
| produce_IfAsn_true b a1 a2 s h st:
  beval s b = true →
  produce (s, h) a1 st →
  produce (s, h) (IfAsn b a1 a2) st
| produce_IfAsn_false b a1 a2 s h st:
  beval s b = false →
  produce (s, h) a2 st →
  produce (s, h) (IfAsn b a1 a2) st
| produce_SepAsn a1 a2 st st' st'':
  produce st a1 st' →
  produce st' a2 st'' →
  produce st (SepAsn a1 a2) st''.

Definition sat0 h s a s' := consume (s, h) a (s', empty_heap).
Inductive sat h s a: Prop :=
| sat_intro s': sat0 h s a s' → sat h s a.

Theorem consume_produce:
  ∀ a h s s' h',
  sat0 h s a s' →
  produce (s, h') a (s', heap_add h h').

Corollary consume_produce_cancel a s h s' h':
  consume (s, h) a (s', h') →
  produce (s, h') a (s', h).
```

7.2 Heap refinement

```
Section PredDefs.

Variable pred_defs: pred_table.

Inductive refines: heap → heap → Prop :=
| refines_refl h:
  refines h h
| refines_add h1 h2 h1' h2':
  refines h1 h1' →
  refines h2 h2' →
  refines (heap_add h1 h2) (heap_add h1' h2')
| refines_pred h0 h p xs vs a:
  refines h0 h →
  pred_defs p = Some (PredDef xs a) →
  length vs = length xs →
  sat h (store_updates store0 xs vs) a →
  refines h0 (singleton_heap (Chunk p vs))
.

Definition is_concrete_heap h :=
  ∀ h' p vs,
  h = heap_add (singleton_heap (Chunk p vs)) h' →
  pred_defs p = None.

Lemma is_concrete_heap_add_l h1 h2:
  is_concrete_heap (heap_add h1 h2) →
  is_concrete_heap h1.

Lemma is_concrete_heap_add h1 h2:
  is_concrete_heap (heap_add h1 h2) →
  is_concrete_heap h1 ∧ is_concrete_heap h2.

Theorem open_theorem h0 h1:
  refines h0 h1 →
  ∀ h p xs vs a,
  h1 = heap_add (singleton_heap (Chunk p vs)) h →
  is_concrete_heap h0 →
  pred_defs p = Some (PredDef xs a) →
  length vs = length xs →
  ∃ h',
  sat h' (store_updates store0 xs vs) a ∧
  refines h0 (heap_add h' h).

Lemma refines_empty:
  ∀ h0 h1,
  refines h0 h1 →
```

```
  h1 = empty_heap →
  h0 = empty_heap.

Lemma refines_heap_add:
  ∀ h0 h1,
  refines h0 h1 →
  ∀ h1a h1b, h1 = heap_add h1a h1b →
  ∃ h0a, ∃ h0b,
  h0 = heap_add h0a h0b ∧
  refines h0a h1a ∧ refines h0b h1b.

Theorem close_theorem:
  ∀ h0 h1 h h' p xs a vs,
  refines h0 h1 →
  h1 = heap_add h h' →
  pred_defs p = Some (PredDef xs a) →
  length vs = length xs →
  sat h (store_updates store0 xs vs) a →
  refines h0 (heap_add (singleton_heap (Chunk p vs)) h').

Lemma refines_concrete h0 h1: refines h0 h1 → is_concrete_heap h1 → h0 = h1.

End PredDefs.
```

Chapter 8

Library ConcreteExecutionFacts

```
Require Export ConcreteExecution.
```

8.1 Facts about specific concrete mutators

```
Definition cupdate_store_(f: store → store): cmutatora store :=
  fun st ⇒ let (s, h) := st in
  single (f s, h) s.
Lemma cwith_store_covers_cupdate_store_ s A (C: cmutatora A):
  cwith_store s C ==> s0 ← cupdate_store_ (fun _ ⇒ s); C;, cupdate_store_ (fun _ ⇒ s0).
Lemma cupdate_store__covers_cwith_store s A (C: cmutatora A):
  s0 ← cupdate_store_ (fun _ ⇒ s); C;, cupdate_store_ (fun _ ⇒ s0) ==> cwith_store s C.
```

Chapter 9

Library SemiconcreteExecutionFacts

Require Export SemiconcreteExecution.

9.1 Facts about specific semiconcrete mutators

9.1.1 with_store, with_store', with_local_store

Definition update_store_ f: scmutatora store :=
  fun st ⇒ let (s, h) := st in single (f s , h) s.

Lemma with_store_covers_update_store s A (C: scmutatora A):
  with_store s C ==> $s0$ ← update_store_ (fun _ ⇒ s); C;, update_store_ (fun _ ⇒ $s0$).

Lemma with_store_covers_update_store' s A (C: scmutatora A):
  $s0$ ← update_store_ (fun _ ⇒ s); C;, update_store_ (fun _ ⇒ $s0$) ==> with_store s C.

Lemma with_store'_covers_update_store s (C: scmutator):
  with_store' s C ==> $s0$ ← update_store_ (fun _ ⇒ s); C; $s1$ ← update_store_ (fun _ ⇒ $s0$); yield $s1$.

Lemma with_store'_covers_update_store' s (C: scmutator):
  $s0$ ← update_store_ (fun _ ⇒ s); C; $s1$ ← update_store_ (fun _ ⇒ $s0$); yield $s1$ ==> with_store' s C.

Lemma with_local_store_covers_update_store A (C: scmutatora A):
  with_local_store C ==> $s0$ ← update_store_ (fun s ⇒ s); C;, update_store_ (fun _ ⇒ $s0$).

Lemma with_local_store_covers_update_store' A (C: scmutatora A):
  $s0$ ← update_store_ (fun s ⇒ s); C;, update_store_ (fun _ ⇒ $s0$) ==> with_local_store C.

Lemma with_store_equiv_update_store s A (C: scmutatora A):
  with_store s C <==> $s0$ ← update_store_ (fun _ ⇒ s); C;, update_store_ (fun _ ⇒ $s0$).

Lemma with_store'_equiv_update_store s (C: scmutator):

with_store' s C <==> $s0$ ← update_store_ (fun _ ⇒ s); C; $s1$ ← update_store_ (fun _ ⇒ $s0$); yield $s1$.

Lemma with_local_store_equiv_update_store A (C: scmutatora A):
with_local_store C <==> $s0$ ← update_store_ (fun s ⇒ s); C;, update_store_ (fun _ ⇒ $s0$).

Lemma with_store_with_store':
∀ s ($C1$ $C2$: scmutator),
with_store s ($C1$; $C2$) ==> s' ← with_store' s $C1$; with_store s' $C2$.

Lemma with_store'_with_store:
∀ s s' (C: scmutator) (f: store → scmutator),
seqf (with_store' s (with_store s' C)) f ==>
with_store s' C; f s.

Lemma with_store_with_store'_ $s0$ s (C: scmutator) B (f: store → scmutatora B):
with_store $s0$ (seqf (with_store' s C) f) ==>
seqf (with_store' s C) (fun $s1$ ⇒ with_store $s0$ (f $s1$)).

Lemma with_store_with_store_ $s0$ s A (C: scmutatora A) B (f: A → scmutatora B):
with_store $s0$ (seqf (with_store s C) f) ==>
seqf (with_store s C) (fun a ⇒ with_store $s0$ (f a)).

Lemma with_store_with_store $s0$ s A (C: scmutatora A):
with_store $s0$ (with_store s C) ==> with_store s C.

9.1.2 set_store

Definition set_store s: scmutatora store :=
fun st ⇒ let ($s0$, h) := st in
single (s, h) $s0$.

Lemma set_store_with_store s A (C: scmutatora A):
$s0$ ← set_store s; x ← C; set_store $s0$; yield x ==> with_store s C.

Lemma set_store_with_store' s (C: scmutator):
$s0$ ← set_store s; C; set_store $s0$ ==> with_store' s C.

Definition get_store: scmutatora store :=
fun st ⇒ let (s, h) := st in
single (s, h) s.

Lemma set_store_with_local_store (C: scmutator):
$s0$ ← get_store; C; set_store $s0$; yield tt ==> with_local_store C.

9.1.3 havoc

Lemma sat_havoc xs s h Q:

sat (havoc xs $(s,\ h)$) $Q \leftrightarrow$
∀ s', (∀ x, ¬ In x xs → s' x = s x) → Q (s', h) tt.

Lemma sat_havoc_elim xs s h Q:
sat (havoc xs $(s,\ h)$) $Q \rightarrow$
∀ s', (∀ x, ¬ In x xs → s' x = s x) → Q (s', h) tt.

Lemma sat_havoc_intro xs s h (Q: state → **unit** → Prop):
(∀ s', (∀ x, ¬ In x xs → s' x = s x) → Q (s', h) tt)
→
sat (havoc xs $(s,\ h)$) Q.

Lemma havoc_covers_noop xs: havoc xs ==> noop.

Chapter 10

Library Locality

```
Require Export SemiconcreteExecutionFacts.
```

10.1 Locality

```
Definition add h: scmutator :=
  fun st ⇒ let (s, h0) := st in
  single (s , heap_add h h0) tt.
Lemma single_mut_add h0: single_mut (add h0).
Definition local {A} (C: scmutatora A) := ∀ h, C;, add h ==> add h; C.
Lemma local_seqf{A B}(C: scmutatora A)(f: A → scmutatora B):
  local C → (∀ x, local (f x)) → local (seqf C f).
Lemma local_seq{A B}(C1: scmutatora A)(C2: scmutatora B):
  local C1 → local C2 → local (C1; C2).
Lemma local_pick_demonic T: local (pick_demonic T).
Lemma local_yield A (x: A): local (yield x:scmutatora A).
Lemma local_iter A (C: scmutatora A) n: local C → local (iter C n).
Lemma local_update_store x v: local (update_store x v).
Lemma local_update_store_n xs vs: local (update_store_n xs vs).
Lemma local_havoc1 x: local (havoc1 x).
Lemma local_noop: local (noop: scmutator).
Lemma local_havoc xs: local (havoc xs).
Lemma local_assume_prop P: local (assume_prop P).
Lemma local_eval_mut e: local (eval_mut e).
Lemma local_evals_mut es: local (evals_mut es).
```

Lemma local_prod_chunk c: local (prod_chunk c).

Lemma local_cons_chunk p ls n: local (cons_chunk p ls n).

Lemma local_cons_chunk_1_1 p l: local (cons_chunk_1_1 p l).

Lemma local_cons_pointsto l: local (cons_pointsto l).

Lemma local_cons_malloc_block l: local (cons_malloc_block l).

Lemma local_cons_pointstos l n: local (cons_pointstos l n).

Lemma local_fork ($C1$ $C2$: scmutator): local $C1$ → local $C2$ → local (fork $C1$ $C2$).

Lemma local_assume_bexpr b: local (assume_bexpr b).

Lemma local_assert_bexpr b: local (assert_bexpr b).

Lemma local_update_store_ f: local (update_store_ f).

Lemma local_fail A: local (fail: scmutatora A).

Lemma local_clear_heap_leak_check (C: scmutator): local (clear_heap; C; leakcheck).

```
Hint Resolve
  local_pick_demonic
  local_iter
  local_update_store
  local_update_store_n
  local_havoc
  local_assume_prop
  local_eval_mut
  local_evals_mut
  local_prod_chunk
  local_cons_chunk
  local_cons_chunk_1_1
  local_cons_pointsto
  local_cons_malloc_block
  local_cons_pointstos
  local_fork
  local_assume_bexpr
  local_noop
  local_assert_bexpr
  local_fail
  local_seqf
  local_seq
  local_update_store_
  local_yield
  local_clear_heap_leak_check
: local.
```

Lemma equiv_local A (C C': scmutatora A): C <==> C' → local C → local C'.

Lemma local_with_store A s (C: scmutatora A): local $C \rightarrow$ local (with_store s C).

Lemma local_with_store' s (C: scmutator): local $C \rightarrow$ local (with_store' s C).

Lemma local_with_local_store A (C: scmutatora A): local $C \rightarrow$ local (with_local_store C).

Hint Resolve local_with_store local_with_store' local_with_local_store: *local*.

Lemma local_prod_pointstos l n: local (prod_pointstos l n).

Hint Resolve local_prod_pointstos: *local*.

Lemma local_consume a: local (consume a).

Lemma local_produce a: local (produce a).

Hint Resolve local_consume local_produce: *local*.

Lemma local_sc_exec *rspecs* *pdefs* c: local (sc_exec *rspecs* *pdefs* c).

Hint Resolve local_sc_exec: *local*.

Chapter 11

Library Modifies

```
Require Export SemiconcreteExecution.
Require Import ListSet.

Local Open Scope list_set_scope.

Definition assert_modified s0 xs: scmutator :=
  fun st ⇒ let (s, h) := st in
  angelicT (fun H: ∀ x, ¬ In x xs → s x = s0 x ⇒ single (s, h) tt).

Definition modifies {A} (C: scmutatora A) xs :=
  ∀ s0,
  assert_modified s0 xs; C ==> C;, assert_modified s0 xs.

Lemma modifies_weaken A (C: scmutatora A) xs1 xs2:
  modifies C xs1 →
  incl xs1 xs2 →
  modifies C xs2.

Lemma update_store_modifies x e: modifies (update_store x e) [x].

Lemma eval_mut_modifies e: modifies (eval_mut e) [].

Lemma pick_demonic_modifies T: modifies (pick_demonic T) [].

Lemma assume_prop_modifies P: modifies (assume_prop P) [].

Lemma prod_chunk_modifies c: modifies (prod_chunk c) [].

Lemma cons_chunk_modifies p ls n: modifies (cons_chunk p ls n) [].

Lemma evals_mut_modifies es: modifies (evals_mut es) [].

Lemma with_store'_modifies s C: modifies (with_store' s C) [].

Lemma with_store_modifies s A (C: scmutatora A): modifies (with_store s C) [].

Lemma with_local_store_modifies A (C: scmutatora A): modifies (with_local_store C) [].

Lemma assume_bexpr_modifies b: modifies (assume_bexpr b) [].

Lemma noop_modifies: modifies noop [].
```

Lemma fail_modifies A xs: modifies (fail: scmutatora A) xs.

Lemma seqf_modifies xs A (C: scmutatora A) B (f: $A \to$ scmutatora B):
modifies C xs $\to$
($\forall$ a, modifies (f a) xs) $\to$
modifies (seqf C f) xs.

Lemma seqf_modifies' $xs1$ $xs2$ xs A (C: scmutatora A) B (f: $A \to$ scmutatora B):
modifies C $xs1$ $\to$
($\forall$ a, modifies (f a) $xs2$) $\to$
incl $xs1$ xs $\to$
incl $xs2$ xs $\to$
modifies (seqf C f) xs.

Lemma seq_modifies xs A ($C1$: scmutatora A) B ($C2$: scmutatora B):
modifies $C1$ xs $\to$
modifies $C2$ xs $\to$
modifies ($C1$; $C2$) xs.

Lemma seq_modifies' $xs1$ $xs2$ xs A ($C1$: scmutatora A) B ($C2$: scmutatora B):
modifies $C1$ $xs1$ $\to$
modifies $C2$ $xs2$ $\to$
incl $xs1$ xs $\to$
incl $xs2$ xs $\to$
modifies ($C1$; $C2$) xs.

Lemma fork_modifies A ($C1$ $C2$: scmutatora A) xs:
modifies $C1$ xs $\to$
modifies $C2$ xs $\to$
modifies (fork $C1$ $C2$) xs.

Lemma fork_modifies' A ($C1$ $C2$: scmutatora A) $xs1$ $xs2$ xs:
modifies $C1$ $xs1$ $\to$
modifies $C2$ $xs2$ $\to$
incl $xs1$ xs $\to$
incl $xs2$ xs $\to$
modifies (fork $C1$ $C2$) xs.

Lemma leakcheck_modifies A (C: scmutatora A): modifies (C ; leakcheck) `[]`.

Lemma leakcheck_modifies' A B ($C1$: scmutatora A) ($C2$: scmutatora B): modifies ($C1$; $C2$; leakcheck) `[]`.

Lemma yield_modifies A (a: A): modifies (yield a) `[]`.

Lemma cons_chunk_1_1_modifies p l: modifies (cons_chunk_1_1 p l) `[]`.

Lemma cons_pointsto_modifies l: modifies (cons_pointsto l) `[]`.

Lemma cons_malloc_block_modifies l: modifies (cons_malloc_block l) `[]`.

`Hint Resolve`

update_store_modifies
eval_mut_modifies
pick_demonic_modifies
assume_prop_modifies
prod_chunk_modifies
cons_chunk_modifies
cons_chunk_1_1_modifies
cons_pointsto_modifies
cons_malloc_block_modifies
evals_mut_modifies
with_store'_modifies
with_store_modifies
with_local_store_modifies
assume_bexpr_modifies
noop_modifies
fail_modifies

seqf_modifies'
seq_modifies'
fork_modifies'

leakcheck_modifies
leakcheck_modifies'

: *modifies*.

Lemma incl_nil A (xs: **list** A): incl [] xs.

Lemma incl_refl A (xs: **list** A): incl xs xs.

Lemma incl_Un_l xs ys: incl xs (xs Un ys).

Lemma incl_Un_r xs ys: incl ys (xs Un ys).

Lemma incl_cons_l A (x: A) xs: incl [x] (x::xs).

Lemma incl_cons_r A (x: A) xs: incl xs (x::xs).

Hint Resolve incl_nil incl_refl incl_Un_l incl_Un_r incl_cons_l incl_cons_r : *modifies*.

Lemma havoc1_modifies x: modifies (havoc1 x) [x].

Hint Resolve havoc1_modifies: *modifies*.

Lemma havoc_modifies xs: modifies (havoc xs) xs.

Hint Resolve havoc_modifies: *modifies*.

Lemma prod_pointstos_modifies l n: modifies (prod_pointstos l n) [].

Hint Resolve prod_pointstos_modifies: *modifies*.

Lemma cons_pointstos_modifies l n: modifies (cons_pointstos l n) [].

Hint Resolve cons_pointstos_modifies: *modifies*.

Lemma sc_exec_modifies *rspecs pdefs c*:
modifies (sc_exec *rspecs pdefs c*) (targets *c*).

Hint Resolve sc_exec_modifies : *modifies*.

Chapter 12

Library SemiconcreteSoundness

```
Require Export Classical.
Require Export ConsumeProduce.
Require Export ConcreteExecutionFacts.
Require Export Locality.
Require Export Modifies.

Local Open Scope nat_scope.

Section Program.

Variable rspecs: spec_table.
Variable pdefs: pred_table.
Variable rdefs: routine_table.

Hypothesis Hvalid_routines: valid_routines rspecs pdefs rdefs.

Hypothesis Hpdefs_pointsto: pdefs pointsto_pred = None.
Hypothesis Hpdefs_malloc_block: pdefs malloc_block_pred = None.
```

12.1 Lifting concrete heaps to semiconcrete heaps

```
Definition cheap_heap(h: cheap): heap :=
  fun c ⇒
  match c with
    Chunk p [l, v] ⇒
    if string_dec p pointsto_pred then
      match h (points_to_id l) with
        None ⇒ 0
      | Some v' ⇒ if Z_eq_dec v v' then 1 else 0
      end
    else if string_dec p malloc_block_pred then
      match h (malloc_block_id l) with
        None ⇒ 0
```

```
      | Some v' ⇒ if Z_eq_dec v v' then 1 else 0
      end
    else
      0
  | _ ⇒ 0
  end.
```

Lemma is_concrete_heap_pointsto l v: is_concrete_heap *pdefs* (singleton_heap (pointsto l v)).

Lemma is_concrete_heap_malloc_block l v: is_concrete_heap *pdefs* (singleton_heap (malloc_block l v)).

Lemma is_concrete_heap_cheap_heap hc: is_concrete_heap *pdefs* (cheap_heap hc).

```
Lemma cheap_heap_update_pointsto hc l v:
  hc (points_to_id l) = None →
  cheap_heap (cheap_update hc (points_to_id l) (Some v)) =
    heap_add (singleton_heap (pointsto l v)) (cheap_heap hc).
```

```
Lemma cheap_heap_update_malloc_block hc l v:
  hc (malloc_block_id l) = None →
  cheap_heap (cheap_update hc (malloc_block_id l) (Some v)) =
    heap_add (singleton_heap (malloc_block l v)) (cheap_heap hc).
```

```
Lemma cheap_heap_add_pointsto hc l v h:
  cheap_heap hc = heap_add (singleton_heap (pointsto l v)) h →
  hc (points_to_id l) = Some v ∧ h = cheap_heap (cheap_update hc (points_to_id l) None).
```

```
Lemma cheap_heap_add_malloc_block hc l v h:
  cheap_heap hc = heap_add (singleton_heap (malloc_block l v)) h →
  hc (malloc_block_id l) = Some v ∧ h = cheap_heap (cheap_update hc (malloc_block_id l) None).
```

12.2 Soundness of a semiconcrete mutator with respect to a concrete mutator

12.2.1 Approximation

```
Definition kappa: mutator state cstate unit :=
  fun st ⇒ let (s, h) := st in
  demonicT (fun hc: {hc | refines pdefs (cheap_heap hc) h} ⇒
    single (s, proj1_sig hc) tt
  ).
```

Lemma demonic_mut_kappa: demonic_mut kappa.

```
Definition approx {A} (Csc: scmutatora A) (Cc: cmutatora A) :=
  Csc;, kappa ==> kappa; Cc.
```

Infix "~~>" := approx (at level 55).

12.2.2 Simple instances

Lemma seq_approx A B ($C1$: scmutatora A) ($C2$: scmutatora B) $C1'$ $C2'$:
$C1$ ~~> $C1'$ → $C2$ ~~> $C2'$ → $C1$; $C2$ ~~> $C1'$; $C2'$.

Lemma seqf_approx A B (C: scmutatora A) (f: A → scmutatora B) C' f':
C ~~> C' → (∀ x, f x ~~> f' x) → seqf C f ~~> seqf C' f'.

Lemma fail_approx A (C: cmutatora A): fail ~~> C.

Lemma approx_block A C: C ~~> (block: cmutatora A).

Lemma pick_demonic_approx: pick_demonic **Z** ~~> pick.

Lemma update_store_approx x v: update_store x v ~~> update_cstore x v.

Lemma assume_prop_approx P b: (P ↔ b = true) → assume_prop P ~~> assumeb b.

Lemma eval_mut_approx e: eval_mut e ~~> ceval_mut e.

Lemma evals_mut_approx es: evals_mut es ~~> cevals_mut es.

Lemma evals_mut_approx' es A (f: **list Z** → scmutatora A) (f': **list Z** → cmutatora A):
(∀ vs, length vs = length es → f vs ~~> f' vs) →
seqf (evals_mut es) f ~~> seqf (cevals_mut es) f'.

Lemma prod_chunk_pointsto_approx l v:
prod_chunk (pointsto l v) ~~> cprod_pointsto l v.

Lemma prod_chunk_malloc_block_approx l v:
prod_chunk (malloc_block l v) ~~> cprod_chunk (malloc_block_id l) v.

Lemma noop_approx: noop ~~> noop.

Lemma cons_pointsto_approx l: cons_pointsto l ~~> ccons_pointsto l.

Lemma cons_malloc_block_approx l: cons_malloc_block l ~~> ccons_malloc_block l.

Lemma fork_approx ($C1$ $C2$: scmutator) $C1'$ $C2'$: $C1$ ~~> $C1'$ → $C2$ ~~> $C2'$ → fork $C1$ $C2$ ~~> fork $C1'$ $C2'$.

Lemma assume_bexpr_approx b: assume_bexpr b ~~> cassume_bexpr b.

Lemma approx_message_mut m: noop ~~> message_mut m.

Lemma covers_approx A ($C1$ $C2$: scmutatora A) $C3$: $C1$ ==> $C2$ → $C2$ ~~> $C3$ → $C1$ ~~> $C3$.

Lemma approx_covers A $C1$ ($C2$ $C3$: cmutatora A): $C2$ ==> $C3$ → $C1$ ~~> $C2$ → $C1$ ~~> $C3$.

Lemma update_store__approx f: update_store_ f ~~> cupdate_store_ f.

Lemma yield_approx A (x: A): yield x ~~> yield x.

Hint Resolve

seq_approx
seqf_approx
fail_approx
approx_block
pick_demonic_approx
update_store_approx
assume_prop_approx
eval_mut_approx
evals_mut_approx
prod_chunk_pointsto_approx
prod_chunk_malloc_block_approx
noop_approx
cons_pointsto_approx
cons_malloc_block_approx
fork_approx
assume_bexpr_approx
approx_message_mut
yield_approx
update_store__approx
: *approx*.

12.2.3 More complex instances

Lemma with_store_approx s A (C: scmutatora A) C': C ~~> C' → with_store s C ~~> cwith_store s C'.

Lemma safe_with_store_leakcheck_noop_covers:
∀ s C,
safe (state0 |> with_store s C; leakcheck) →
local C →
(noop: scmutator) ==> with_store s C.

Lemma consume_soundness:
∀ a s h Q,
sat (consume a (s, h)) Q →
∃ s' h',
ConsumeProduce.consume (s, h) a (s', h') ∧ Q (s', h') tt.

Lemma produce_soundness:
∀ a s h Q,
sat (produce a (s, h)) Q →
∀ s' h',
ConsumeProduce.produce (s, h) a (s', h') → Q (s', h') tt.

Lemma consume_produce A s a S (f: store → store → state → **outcome** S A):

```
  s1 ← with_store' s (consume a);
  s2 ← with_store' s (produce a);
  f s1 s2
  ==>
  s' ← pick_angelic store;
  f s' s'.

Lemma consume_produce0 s a S A (C: state → outcome S A):
  with_store s (consume a);
  with_store s (produce a);
  C
  ==>
  C.

Lemma routine_call_soundness:
  ∀
    n
    (IHn: ∀ c, sc_exec rspecs pdefs c ~~> c_exec_n rdefs n c)
    x es xs' pre post xs c
    (Hlength: length es = length xs')
    (Hvalid_routine:
     safe (
       state0 |>
       vs ← iter (pick_demonic Z) (length xs');
       with_store store0 (
         s1 ← with_store' (store_updates store0 xs' vs) (produce pre);
         v ← with_store (store_updates store0 xs vs) (sc_exec rspecs pdefs c; eval_mut
result);
         with_store (store_update s1 result v) (consume post)
       );
       leakcheck
     )),
  vs ← evals_mut es;
  s1 ← with_store' (store_updates store0 xs' vs) (consume pre);
  v ← pick_demonic Z;
  with_store (store_update s1 result v) (produce post);
  update_store x v
  ~~>
  vs ← cevals_mut es;
  v ← cwith_store (store_updates store0 xs vs) (
    c_exec_n rdefs n c;
    ceval_mut result
  );
  update_cstore x v.
```

```
Lemma open_soundness p es xs a:
  pdefs p = Some (PredDef xs a) →
  length es ≤ length xs →
  ls ← evals_mut es;
  vs ← cons_chunk p ls (length xs - length es);
  with_store (store_updates store0 xs (ls ++ vs)) (produce a)
  ~~>
  noop.

Lemma close_soundness p es xs a:
  pdefs p = Some (PredDef xs a) →
  length es = length xs →
  vs ← evals_mut es;
  with_store (store_updates store0 xs vs) (consume a);
  prod_chunk (Chunk p vs) ~~> noop.

Lemma prod_pointstos_approx l n: prod_pointstos l n ~~> cprod_pointstos l n.

Hint Resolve prod_pointstos_approx: approx.

Lemma cons_pointstos_approx l n: cons_pointstos l n ~~> ccons_pointstos l n.

Hint Resolve cons_pointstos_approx: approx.
```

12.2.4 Loops

```
Definition sc_exec := sc_exec rspecs pdefs.

Lemma assert_modified_covers S1 A (C1 C2: mutator state S1 A) xs:
  (∀ s0, assert_modified s0 xs; C1 ==> C2) →
  C1 ==> C2.

Lemma local_sat (C: scmutator) s h Q:
  local C →
  sat (C (s, empty_heap)) Q →
  (∀ s h, Q (s, h) tt → h = empty_heap) →
  sat (C (s, h)) (fun st' _ ⇒ let (s', h') := st' in h' = h ∧ Q (s', empty_heap) tt).

Lemma modifies_sat C xs s h Q:
  modifies C xs →
  sat (C (s, h)) Q →
  sat (C (s, h)) (fun st' _ ⇒ let (s', h') := st' in (∀ x, ¬ In x xs → s' x = s x) ∧ Q st'
tt).

Lemma while_case_split (C: scmutator) xs s1 a:
  modifies C xs →
  local C →
  assert_modified s1 xs;
  with_local_store (consume a);
```

```
  havoc xs;
  clear_heap;
  C;
  leakcheck
  ==> fail
  ∨
  assert_modified s1 xs;
  havoc xs
  ==> C.
```

Lemma pick_approx: pick ~~> pick.

Lemma iterate_n_approx *n* *C* *C'*: *C* ~~> *C'* → iterate_n *n* *C* ~~> iterate_n *n* *C'*.

Lemma iterate_approx (*C*: scmutator) (*C'*: cmutator): *C* ~~> *C'* → iterate *C* ~~> iterate *C'*.

Lemma covers_iterate *C1* (*C2*: scmutator):
  C1 ==> noop →
  C1 ==> *C2*; *C1* →
  C1 ==> iterate *C2*.

Lemma assert_modified_covers_noop *s* *xs*: assert_modified *s* *xs* ==> noop.

Lemma with_local_store_consume_produce *a*:
  with_local_store (consume *a*); with_local_store (produce *a*) ==> noop.

Lemma assert_modified_dup *s* *xs*:
  assert_modified *s* *xs* ==> assert_modified *s* *xs*; assert_modified *s* *xs*.

Lemma havoc_dup *xs*: havoc *xs* ==> havoc *xs*; havoc *xs*.

Lemma while_soundness *n* *b* *a* *c*
  (*IHn*: sc_exec *c* ~~> c_exec_n *rdefs* *n* *c*):
  sc_exec (While *b* *a* *c*) ~~> c_exec_n *rdefs* (S *n*) (While *b* *a* *c*).

12.2.5 Toplevel lemmas and theorem

Lemma main_lemma0 *n*: ∀ *c*, sc_exec *c* ~~> c_exec_n *rdefs* *n* *c*.

Lemma approx_pick *A* (*C*: scmutatora *A*) *f*: (∀ *x*, *C* ~~> *f* *x*) → *C* ~~> seqf pick *f*.

Lemma main_lemma *c*: sc_exec *c* ~~> c_exec *rdefs* *c*.

Lemma approx_safe (*C1*: scmutator) *C2*: *C1* ~~> *C2* → safe (*C1* state0) → safe (*C2* cstate0).

End Program.

Theorem semiconcrete_soundness *rspecs* *pdefs* *rdefs* *c*:
  valid_program *rspecs* *pdefs* *rdefs* *c* →
  cvalid_program *rdefs* *c*.

Chapter 13

Library SMT

```
Require Export List.
Require Export ZArith.
```

13.1 Symbols, terms, and formulae

```
Definition symbol := nat.

Inductive term :=
| t_lit: Z → term
| t_symb: symbol → term
| t_add: term → term → term
.

Inductive formula :=
| f_eq: term → term → formula
| f_lt: term → term → formula
| f_and: formula → formula → formula
| f_not: formula → formula.

Definition f_true := f_eq (t_lit 0) (t_lit 0).
```

13.2 A minimal SMT solver

```
Module Solver.

Definition result0 := option (list formula).

Definition term_dec(t1 t2: term): {t1 = t2} + {t1 ≠ t2}.
Defined.

Definition formula_dec(f1 f2: formula): {f1 = f2} + {f1 ≠ f2}.
Defined.
```

```
Definition is_fact f facts := in_dec formula_dec f facts.

Definition assume_neq t1 t2 facts :=
  if term_dec t1 t2 then None else
  if is_fact (f_eq t1 t2) facts then None else
  Some (f_not (f_eq t1 t2)::f_not (f_eq t2 t1)::facts).

Definition seq0 op1 (op2: list formula → result0) (facts: list formula) :=
  match op1 facts with
    None ⇒ None
  | Some facts ⇒ op2 facts
  end.

Fixpoint assume f facts :=
  if is_fact (f_not f) facts then None else
  match f with
    f_eq t1 t2 ⇒
    Some (f_eq t1 t2::f_eq t2 t1::facts)
  | f_lt t1 t2 ⇒
    seq0
      (assume_neq t1 t2)
      (fun facts ⇒ Some (f::facts))
      facts
  | f_and f1 f2 ⇒
    seq0
      (assume f1)
      (assume f2)
      facts
  | f_not f ⇒ assume_false f facts
  end
with assume_false f facts :=
  if is_fact f facts then None else
  match f with
    f_eq t1 t2 ⇒
    if term_dec t1 t2 then None else
    Some (f_not f::facts)
  | f_lt t1 t2 ⇒
    Some (f_not f::facts)
  | f_and f1 f2 ⇒
    Some (f_not f::facts)
  | f_not f ⇒ assume f facts
  end.

Inductive result := valid | undecided.

Definition solver(f: formula): result :=
```

```
  match assume_false f nil with
    None ⇒ valid
  | _ ⇒ undecided
  end.
End Solver.
Definition solver := Solver.solver.
Definition follows(pc: formula)(goal: formula) :=
  if solver (f_not (f_and pc (f_not goal))) then true else false.
```

13.3 Free symbols

```
Fixpoint term_symbols t :=
  match t with
    t_lit z ⇒ nil
  | t_symb s ⇒ s::nil
  | t_add t1 t2 ⇒ term_symbols t1 ++ term_symbols t2
  end.
Fixpoint formula_symbols f :=
  match f with
    f_eq t1 t2 ⇒ term_symbols t1 ++ term_symbols t2
  | f_lt t1 t2 ⇒ term_symbols t1 ++ term_symbols t2
  | f_and f1 f2 ⇒ formula_symbols f1 ++ formula_symbols f2
  | f_not f ⇒ formula_symbols f
  end.
Definition fresh_symbol ss :=
  fold_left (fun m s ⇒ max m (S s)) ss 0.
```

Chapter 14

Library SymbolicExecution

```
Require Export ClassicalEpsilon.
Require Export Bool.
Require Export SetoidDec.
Require Export SMT.
Require Export Programs.
Require Import String.
Require Export List.
Require Export Outcomes.
```

14.1 Symbolic states

```
Definition sstore := StringMap.t term.
Inductive schunk := SChunk(p: pred)(ts: list term).
Definition spointsto l v := SChunk pointsto_pred [l, v].
Definition smalloc_block l v := SChunk malloc_block_pred [l, v].
Definition sheap := list schunk.
Inductive sstate := SState(pc: formula)(ss: sstore)(sh: sheap).

Definition sstore0: sstore := StringMap.empty _.
Definition sstate0 := SState f_true sstore0 nil.
```

14.2 Symbolic evaluation

```
Definition sstore_lookup s x :=
  match StringMap.find x s with Some t ⇒ t | None ⇒ t_lit 0 end.
Definition sstore_update (s: sstore) x t := StringMap.add x t s.
Fixpoint sstore_updates (s: sstore) xs ts :=
  match xs, ts with
    x::xs, t::ts ⇒ sstore_updates (sstore_update s x t) xs ts
```

```
  | _, _ ⇒ s
  end.

Fixpoint seval(s: sstore)(e: expr) :=
  match e with
    e_lit z ⇒ t_lit z
  | e_var x ⇒ sstore_lookup s x
  | e_add e1 e2 ⇒ t_add (seval s e1) (seval s e2)
  end.

Fixpoint sbeval(s: sstore)(b: bexpr) :=
  match b with
    b_eq e1 e2 ⇒ f_eq (seval s e1) (seval s e2)
  | b_lt e1 e2 ⇒ f_lt (seval s e1) (seval s e2)
  | b_not b ⇒ f_not (sbeval s b)
  end.
```

14.3 Symbolic outcomes

```
Definition soutcome A := outcome sstate A.
Definition smutatora A := sstate → soutcome A.
Definition smutator := smutatora unit.
```

14.4 Symbolic basic mutators

```
Definition fresh_mut: smutatora term :=
  fun st ⇒ let (pc, s, h) := st in
  let t := t_symb (fresh_symbol (formula_symbols pc)) in
  single (SState (f_and pc (f_eq t t)) s h) t.

Definition assume_formula(phi: formula): smutator :=
  fun st ⇒ let (pc, s, h) := st in
  if follows pc (f_not phi) then
    oblock
  else
    single (SState (f_and pc phi) s h) tt.

Definition update_sstore(x: var)(t: term): smutator :=
  fun st ⇒ let (ph, s, h) := st in
  single (SState ph (sstore_update s x t) h) tt.

Definition update_sstore_n(xs: list var)(ts: list term): smutator :=
  fun st ⇒ let (ph, s, h) := st in
  single (SState ph (sstore_updates s xs ts) h) tt.
```

```
Definition shavoc1(x: var): smutator :=
  v ← fresh_mut;
  update_sstore x v.

Fixpoint shavoc(xs: list var): smutator :=
  match xs with
    [] ⇒ noop
  | x::xs ⇒ shavoc1 x; shavoc xs
  end.

Definition swith_store{A}(s: sstore)(op: smutatora A): smutatora A :=
  fun st ⇒ let (pc, s0, h) := st in
  bindf
    (op (SState pc s h))
    (fun a st ⇒ let (pc, _, h) := st in single (SState pc s0 h) a).

Definition swith_store'(s: sstore)(op: smutator): smutatora sstore :=
  fun st ⇒ let (pc, s0, h) := st in
  bindf
    (op (SState pc s h))
    (fun _ st ⇒ let (pc, s, h) := st in single (SState pc s0 h) s).

Definition swith_local_store(op: smutator): smutator :=
  fun st ⇒ let (pc, s0, h) := st in
  bindf
    (op (SState pc s0 h))
    (fun _ st ⇒ let (pc, s, h) := st in single (SState pc s0 h) tt).

Definition seval_mut(e: expr): smutatora term :=
  fun st ⇒ let (pc, s, h) := st in
  single st (seval s e).

Definition sevals_mut(es: list expr) :=
  fun st: sstate ⇒ let (pc, s, h) := st in
  single st (map (seval s) es).

Definition sbeval_mut(b: bexpr): smutatora formula :=
  fun st ⇒ let (pc, s, h) := st in
  single st (sbeval s b).

Definition sassume_bexpr(b: bexpr): smutator :=
  seqf (sbeval_mut b) assume_formula.

Definition sassert_bexpr(b: bexpr): smutator :=
  fun st ⇒ let (pc, s, h) := st in
  if follows pc (sbeval s b) then
    single st tt
  else
    ofail.
```

```
Definition sclear_heap: smutator :=
  fun st ⇒ let (pc, s, h) := st in
  single (SState pc s nil) tt.

Definition sleakcheck: smutator :=
  fun st ⇒ let (pc, s, h) := st in
  match h with
    nil ⇒ oblock
  | SChunk p _::_ ⇒ message ("Leaking chunk " ++ p) ofail
  end.

Fixpoint extract0{A}{B}(f: A → option B)(todo done: list A): option (list A × B) :=
  match todo with
    nil ⇒ None
  | x::xs ⇒
    match f x with
      None ⇒ extract0 f xs (x::done)
    | Some y ⇒ Some (done ++ xs, y)
    end
  end.

Definition extract{A B}(f: A → option B)(xs: list A): option (list A × B) :=
  extract0 f xs nil.

Fixpoint forall2b{A B}(xs: list A)(ys: list B)(f: A → B → bool): bool :=
  match xs, ys with
    x::xs, y::ys ⇒ f x y && forall2b xs ys f
  | _, _ ⇒ true
  end.

Definition match_chunk pc p ts n (c: schunk) :=
  let (p', ts') := c in
  if string_dec p p' then
    if (length ts' == length ts + n)%nat then
      if forall2b ts (firstn (length ts) ts') (fun t t' ⇒ follows pc (f_eq t t')) then
        Some (skipn (length ts) ts')
      else
        None
    else
      None
  else
    None.

Definition extract_chunk p ts n: smutatora (list term) :=
  fun st ⇒ let (pc, s, h) := st in
  match extract (match_chunk pc p ts n) h with
    None ⇒ ofail
```

```
  | Some (h, ts') ⇒ single (SState pc s h) ts'
  end.

Definition extract_chunk_1_1 p l :=
  ts ← extract_chunk p [l] 1;
  yield (hd (t_lit 0) ts).

Definition extract_pointsto l := extract_chunk_1_1 pointsto_pred l.
Definition extract_malloc_block l := extract_chunk_1_1 malloc_block_pred l.

Fixpoint extract_pointstos l n :=
  match n with
    O ⇒ noop
  | S n ⇒
    extract_pointsto l;
    extract_pointstos (t_add l (t_lit 1)) n
  end.

Definition sprod_chunk(c: schunk): smutator :=
  fun st ⇒ let (pc, s, h) := st in
  single (SState pc s (c::h)) tt.

Fixpoint sprod_pointstos l n: smutator :=
  match n with
    O ⇒ noop
  | S n ⇒
    v ← fresh_mut;
    sprod_chunk (spointsto l v);
    sprod_pointstos (t_add l (t_lit 1)) n
  end.

Definition constant_term_value t: smutatora Z :=
  match t with
    t_lit z ⇒ yield z
  | _ ⇒ fail
  end.
```

14.5 Symbolic execution

```
Fixpoint sconsume(a: asn): smutator :=
  match a with
    BAsn b ⇒ sassert_bexpr b
  | PAsn p es xs ⇒
    message_mut "sconsume PAsn";
    ts ← sevals_mut es;
    ts' ← extract_chunk p ts (length xs);
```

```
    update_sstore_n xs ts'
  | IfAsn b a1 a2 ⇒
    fork (
      sassume_bexpr b;
      message_mut "sconsume IfAsn true";
      sconsume a1
    ) (
      sassume_bexpr (b_not b);
      message_mut "sconsume IfAsn false";
      sconsume a2
    )
  | SepAsn a1 a2 ⇒
    message_mut "sconsume SepAsn a1";
    sconsume a1;
    message_mut "sconsume SepAsn a2";
    sconsume a2
  end.

Fixpoint sproduce(a: asn): smutator :=
  match a with
    BAsn b ⇒ sassume_bexpr b
  | PAsn p es xs ⇒
    ts ← sevals_mut es;
    ts' ← iter fresh_mut (length xs);
    update_sstore_n xs ts';
    sprod_chunk (SChunk p (ts ++ ts'))
  | IfAsn b a1 a2 ⇒
    fork (
      sassume_bexpr b;
      sproduce a1
    ) (
      sassume_bexpr (b_not b);
      sproduce a2
    )
  | SepAsn a1 a2 ⇒
    sproduce a1;
    sproduce a2
  end.

Section SymbolicExecution.

Variable routine_specs: spec_table.
Variable pred_defs: pred_table.

Fixpoint s_exec(c: cmd): smutator :=
```

```
match c with
| Assign x e ⇒
  t ← seval_mut e;
  update_sstore x t
| Malloc x n ⇒
  message_mut "Malloc";
  l ← fresh_mut;
  update_sstore x l;
  assume_formula (f_lt (t_lit 0) l);
  sprod_chunk (smalloc_block l (t_lit (Z.of_nat n)));
  sprod_pointstos l n
| Free e ⇒
  message_mut "Free";
  tl ← seval_mut e;
  tn ← extract_malloc_block tl;
  n ← constant_term_value tn;
  extract_pointstos tl (Z.to_nat n)
| Read x e ⇒
  message_mut "Read";
  t1 ← seval_mut e;
  t2 ← extract_pointsto t1;
  sprod_chunk (spointsto t1 t2);
  update_sstore x t2
| Write e1 e2 ⇒
  message_mut "Write";
  t1 ← seval_mut e1;
  t2 ← seval_mut e2;
  _ ← extract_pointsto t1;
  sprod_chunk (spointsto t1 t2)
| Call x r es ⇒
  message_mut "Call";
  match StringMap.find r routine_specs with
    None ⇒ fail
  | Some (RoutineSpec xs pre post) ⇒
    if length es == length xs then
      ts ← sevals_mut es;
      s1 ← swith_store' (sstore_updates sstore0 xs ts) (sconsume pre);
      t ← fresh_mut;
      swith_store (sstore_update s1 result t) (sproduce post);
      update_sstore x t
    else
      fail
```

```
  end
| IfCmd b c1 c2 ⇒
  fork (
    sassume_bexpr b;
    message_mut "IfCmd true";
    s_exec c1
  ) (
    sassume_bexpr (b_not b);
    message_mut "IfCmd false";
    s_exec c2
  )
| While b a c ⇒
  message_mut "While";
  swith_local_store (sconsume a);
  shavoc (targets c);
  fork (
    sclear_heap; (
      swith_local_store (sproduce a);
      sassume_bexpr b;
      s_exec c;
      swith_local_store (sconsume a)
    );
    sleakcheck
  ) (
    swith_local_store (sproduce a);
    sassume_bexpr (b_not b)
  )
| Seq c1 c2 ⇒
  s_exec c1;
  s_exec c2
| Open p es ⇒
  message_mut "Open";
  match pred_defs p with
    None ⇒ fail
  | Some (PredDef xs a) ⇒
    if le_dec (length es) (length xs) then
      ts1 ← sevals_mut es;
      ts2 ← extract_chunk p ts1 (length xs - length es);
      swith_store (sstore_updates sstore0 xs (ts1 ++ ts2)) (
        sproduce a
      )
    else
```

```
        fail
      end
    | Close p es ⇒
      message_mut "Close";
      match pred_defs p with
        None ⇒ fail
      | Some (PredDef xs a) ⇒
        if length es == length xs then
          ts ← sevals_mut es;
          swith_store (sstore_updates sstore0 xs ts) (
            sconsume a
          );
          sprod_chunk (SChunk p ts)
        else
          fail
      end
    | Skip ⇒
      message_mut "Skip"
    | Message m ⇒
      message_mut m
    end.

Inductive eresult A := ok | error: A → eresult A.

Global Implicit Arguments ok [A].
Global Implicit Arguments error [A].

Fixpoint eresult_bind {A} (r: eresult A) (op: eresult A): eresult A :=
  match r with
    ok ⇒ op
  | error msg ⇒ error msg
  end.

Notation "op1 &&> op2" := (eresult_bind op1 op2) (at level 90).

Definition eresult_annotate(msg: string)(r: eresult string): eresult string :=
  match r with
    ok ⇒ ok
  | error msg' ⇒ error (msg ++ msg')%string
  end.

Fixpoint sresult_ok0{A}(prefix: string)(r: soutcome A): eresult string :=
  match r with
    single st a ⇒ ok
  | demonic (set_ n_Empty_set _) ⇒ ok
  | demonic (set_ n_bool o) ⇒ sresult_ok0 prefix (o true) &&> sresult_ok0 prefix (o false)
  | demonic _ ⇒ if excluded_middle_informative (safe r) then ok else error prefix
```

```
  | angelic (set_ n_Empty_set _) ⇒ error (prefix ++ " -> " ++ "failure")%string
  | angelic (set_ n_bool o) ⇒ error (prefix ++ " -> " ++ "angelic fork")%string
  | angelic _ ⇒ if excluded_middle_informative (safe r) then ok else error prefix
  | message msg o ⇒ sresult_ok0 (prefix ++ "; " ++ msg)%string o
  end.
Definition sresult_ok{A} := sresult_ok0 (A:=A) ""%string.
Section SRoutineDefs.
Variable rdefs: routine_table.
Definition svalid_routine r (rspec: routine_spec) :=
  eresult_annotate ("Routine " ++ r ++ ": ")%string (
  match rdefs r with
    None ⇒ error "routine not implemented"%string
  | Some (RoutineDef xs c) ⇒
    let (xs', pre, post) := rspec in
    sresult_ok (
      sstate0 |>
      vs ← iter fresh_mut (length xs');
      swith_store sstore0 (
        s1 ← swith_store' (sstore_updates sstore0 xs' vs) (sproduce pre);
        t ← swith_store (sstore_updates sstore0 xs vs) (s_exec c; seval_mut result);
        message_mut "Checking postcondition";
        swith_store (sstore_update s1 result t) (sconsume post)
      );
      sleakcheck
    )
  end
  ).
Fixpoint foralle{A B}(f: A → eresult B)(xs: list A): eresult B :=
  match xs with
    nil ⇒ ok
  | x::xs ⇒ f x &&> foralle f xs
  end.
Definition svalid_routines :=
  foralle
    (fun el ⇒ svalid_routine (fst el) (snd el))
    (StringMap.elements routine_specs).
Definition svalid_command c :=
  sresult_ok (
    sstate0 |> s_exec c
  ).
Definition pdefs_check :=
```

```
  match pred_defs pointsto_pred with
    None ⇒ ok
  | _ ⇒ error "|-> should not be user-defined"%string
  end &&>
  match pred_defs malloc_block_pred with
    None ⇒ ok
  | _ ⇒ error "mb should not be user-defined"%string
  end.

Definition svalid_program c :=
  pdefs_check &&>
  svalid_routines &&>
  svalid_command c.

End SRoutineDefs.

End SymbolicExecution.
```

14.6 Example symbolic executions

```
Open Local Scope string_scope.

Local Notation l := "l".
Local Notation v := "v".
Local Notation next := "next".
Local Notation list := "list".
Local Notation mb := malloc_block_pred.
Local Notation n := "n".

Definition list_pred := "list".
Definition list_pred_def :=
  PredDef [l] (
    If l == 0 Then
      b_true
    Else (
      #mb(l, 2) &*& l |-> _ &*& (l + 1) |-> ??next &*& #list(next)
    )
  ).

Local Notation a := "a".
Local Notation b := "b".
Local Notation tail := "tail".

Definition make_range_spec :=
  RoutineSpec [a, b]
    (b_true)
    (#list(result)).
```

```
Definition make_range :=
  RoutineDef [a, b] (
    If a == b Then (
      result ':=' 0
    ) Else (
      next ':=' call "make_range"(a + 1, b);
      result ':=' malloc(2); [result] ':=' a; [result + 1] ':=' next
    );
    close list(result)
  ).

Definition reverse_spec :=
  RoutineSpec [a, b]
    (#list(a) &*& #list(b))
    (#list(result)).

Definition reverse :=
  RoutineDef [a, b] (
    open list(a);
    If a == 0 Then
      result ':=' b
    Else (
      next ':=' [a + 1];
      [a + 1] ':=' b;
      close list(a);
      result ':=' call "reverse"(next, a)
    )
  ).

Definition dispose_spec :=
  RoutineSpec [a]
    (#list(a))
    (b_true).

Definition dispose :=
  RoutineDef [a] (
    open list(a);
    If a == 0 Then
      Skip
    Else (
      next ':=' [a + 1];
      free(a);
      call "dispose"(next)
    )
  ).
```

```
Definition reverse_loop_spec :=
  RoutineSpec [a]
    (#list(a))
    (#list(result)).

Definition reverse_loop :=
  RoutineDef [a] (
    close list(b);
    while ¬ (a == 0) invariant #list(a) &*& #list(b) do (
      open list(a);
      tail ':=' [a + 1];
      [a + 1] ':=' b;
      b ':=' a;
      a ':=' tail;
      close list(b)
    );
    open list(a);
    result ':=' b
  ).

Definition dispose_loop :=
  RoutineDef [a] (
    while ¬ (a == 0) invariant #list(a) do (
      open list(a);
      tail ':=' [a + 1];
      free(a);
      a ':=' tail
    );
    open list(a)
  ).

Definition my_rspecs :=
  StringMap.empty _
    |> StringMap.add "make_range" make_range_spec
    |> StringMap.add "reverse" reverse_spec
    |> StringMap.add "reverse_loop" reverse_loop_spec
    |> StringMap.add "dispose" dispose_spec
    |> StringMap.add "dispose_loop" dispose_spec
    .

Definition my_rdefs :=
  (fun r ⇒
     if string_dec r "make_range" then Some make_range else
     if string_dec r "reverse" then Some reverse else
     if string_dec r "reverse_loop" then Some reverse_loop else
```

```
    if string_dec r "dispose" then Some dispose else
    if string_dec r "dispose_loop" then Some dispose_loop else
    None).
Definition my_preddefs :=
  fun p ⇒
  if string_dec p list_pred then Some list_pred_def else
  None.
Definition reverse_test :=
  (
    a ':=' call "make_range"(0, 5);
    close list(0);
    b ':=' call "reverse"(a, 0);
    close list(0);
    a ':=' call "reverse"(b, 0);
    call "dispose"(a)
  )%cmd.
Goal
  svalid_command my_rspecs my_preddefs reverse_test = ok.
Goal
  svalid_routine my_rspecs my_preddefs my_rdefs "make_range" make_range_spec = ok.
Goal
  svalid_program my_rspecs my_preddefs my_rdefs reverse_test = ok.
Definition reverse_test_broken :=
  (
    a ':=' call "make_range"(0, 5);
    close list(0, 0);
    b ':=' call "reverse"(a, 0);
    close list(0, 0);
    call "reverse"(b + 10, 0)
  )%cmd.
Goal
  svalid_program my_rspecs my_preddefs my_rdefs reverse_test_broken ≠ ok.
Definition reverse_loop_test :=
  (
    a ':=' call "make_range"(0, 5);
    b ':=' call "reverse_loop"(a);
    call "reverse_loop"(b)
  )%cmd.
Goal
  svalid_program my_rspecs my_preddefs my_rdefs reverse_loop_test = ok.
```

Chapter 15

Library SMTSoundness

```
Require Export Classical.
Require Export SMT.
```

15.1 Interpreting terms and formulae

```
Definition interp := symbol → Z.

Section Interpretation.

Variable I: interp.

Fixpoint term_interp(t: term): Z :=
  match t with
    t_lit z ⇒ z
  | t_symb s ⇒ I s
  | t_add t1 t2 ⇒ (term_interp t1 + term_interp t2)%Z
  end.

Fixpoint fsat(f: formula): Prop :=
  match f with
    f_eq t1 t2 ⇒ term_interp t1 = term_interp t2
  | f_lt t1 t2 ⇒ (term_interp t1 < term_interp t2)%Z
  | f_and f1 f2 ⇒ fsat f1 ∧ fsat f2
  | f_not f ⇒ ¬ fsat f
  end.
```

15.2 Soundness of the SMT solver

```
Import Solver.

Definition fssat facts := ∀ f, In f facts → fsat f.
```

```
Fixpoint r0sat r0 :=
  match r0 with
    None ⇒ False
  | Some fs ⇒ fssat fs
  end.
```

Lemma assume_neq_sound *facts t1 t2*:
  fssat *facts* → term_interp *t1* ≠ term_interp *t2* →
  r0sat (assume_neq *t1 t2 facts*).

Lemma seq0_sound *op1 op2 facts*:
  r0sat (*op1 facts*) →
  (∀ *facts'*, fssat *facts'* → r0sat (*op2 facts'*)) →
  r0sat (seq0 *op1 op2 facts*).

Lemma assume_sound *f*:
  ∀ *facts*,
  (fsat *f* → fssat *facts* → r0sat (assume *f facts*)) ∧
  (¬ fsat *f* → fssat *facts* → r0sat (assume_false *f facts*)).

Lemma solver_sound *f*: solver *f* = valid → fsat *f*.

Lemma follows_sound *pc f*:
  follows *pc f* = true → fsat *pc* → fsat *f*.

End Interpretation.

15.3 Free symbols: properties

Lemma fold_left_max_S *ss*: ∀ *x*, *x* ≤ fold_left (fun *m s* ⇒ max *m* (S *s*)) *ss x*.

Lemma In_lt_fold_left_max_S *ss*: ∀ *x y*, In *x ss* → *x* < fold_left (fun *m s* ⇒ max *m* (S *s*)) *ss y*.

Lemma fresh_symbol_sound *ss*:
  ¬ In (fresh_symbol *ss*) *ss*.

Lemma term_symbols_sound *I I' t*:
  (∀ *s*, In *s* (term_symbols *t*) → *I s* = *I' s*) →
  term_interp *I t* = term_interp *I' t*.

Lemma formula_symbols_sound *f*:
  ∀ *I I'*,
  (∀ *s*, In *s* (formula_symbols *f*) → *I s* = *I' s*) →
  fsat *I f* → fsat *I' f*.

Chapter 16

Library SymbolicExecutionFacts

```
Require Export SymbolicExecution.
```

16.1 Generic facts about symbolic mutators

```
Lemma sstore_lookup_sstore_update_eq ss x t:
  sstore_lookup (sstore_update ss x t) x = t.

Lemma sstore_lookup_sstore_update_neq ss x t y:
  x ≠ y → sstore_lookup (sstore_update ss x t) y = sstore_lookup ss y.

Definition set_sstore ss: smutatora sstore :=
  fun st ⇒ let (pc, ss0, sh) := st in
  single (SState pc ss sh) ss0.

Lemma swith_store_set_sstore ss A (C: smutatora A):
  swith_store ss C ==> ss0 ← set_sstore ss; x ← C; set_sstore ss0; yield x.

Lemma swith_store'_set_sstore ss (C: smutator):
  swith_store' ss C ==> ss0 ← set_sstore ss; C; set_sstore ss0.

Definition get_sstore: smutatora sstore :=
  fun st ⇒ let (pc, ss, sh) := st in
  single st ss.

Lemma swith_local_store_set_sstore (C: smutator):
  swith_local_store C ==> ss0 ← get_sstore; C; set_sstore ss0; yield tt.
```

Chapter 17

Library EnsemblesEx

Require Export List.
Require Export Ensembles.
Require Export SetoidClass.

17.1 Ensembles (sets as predicates)

Notation "A 'Un' B" := (**Union** _ A B) (at level 51, right associativity).
Notation "{| x |}" := (**Singleton** _ x).
Notation "A <_B" := (Included _ A B) (at level 70).
Notation "x :_A" := (In _ A x) (at level 70).
Notation "{}" := (**Empty_set** _).

Lemma Same_set_equivalence A: **Equivalence** (Same_set A).

Program Instance ensemble_setoid A : **Setoid** (Ensemble A) :=
{ equiv := Same_set _ ; setoid_equiv := Same_set_equivalence _ }.

Lemma Union_commut A (X Y: Ensemble A): X Un Y = Y Un X.

Lemma Union_assoc A (X Y Z: Ensemble A): (X Un Y) Un Z = X Un Y Un Z.

Lemma Union_idem A (X: Ensemble A): X Un X = X.

Lemma Union_Empty_set A (X: Ensemble A): X Un {} = X.

Lemma Union_Includes_iff A (X Y Z: Ensemble A): X Un Y <_ Z $\leftrightarrow$ X <_ Z $\wedge$ Y <_ Z.

Lemma Union_Includes_intro A (X Y Z: Ensemble A): X <_ Z $\rightarrow$ Y <_ Z $\rightarrow$ X Un Y <_ Z.

Lemma Union_absorb_r A (X Y: Ensemble A): Y <_ X $\rightarrow$ X Un Y = X.

Definition elems {A} (xs: **list** A): Ensemble A := fun x $\Rightarrow$ List.In x xs.

Lemma elems_nil A: elems (nil: **list** A) = **Empty_set** A.

Lemma elems_cons A (x: A) xs: elems (x::xs) = {|x|} Un elems xs.

Lemma elems_app {A} (xs ys: **list** A): elems (xs ++ ys) = elems xs Un elems ys.

Chapter 18

Library PartialInterpretations

```
Require Export Bool.
Require Export EnsemblesEx.
Require Export SMTSoundness.
```

18.1 Partial interpretations

```
Definition pinterp := symbol → option Z.
Inductive term_pinterp(I: pinterp): term → Z → Prop :=
| term_pinterp_t_lit z: term_pinterp I (t_lit z) z
| term_pinterp_t_symb s z: I s = Some z → term_pinterp I (t_symb s) z
| term_pinterp_t_add t1 t2 z1 z2:
  term_pinterp I t1 z1 → term_pinterp I t2 z2 → term_pinterp I (t_add t1 t2) (z1 +
z2)%Z.
Inductive fsatp(I: pinterp): formula → bool → Prop :=
| fsatp_f_eq t1 t2 z1 z2:
  term_pinterp I t1 z1 → term_pinterp I t2 z2 → fsatp I (f_eq t1 t2) (if Z_eq_dec z1
z2 then true else false)
| fsatp_f_lt t1 t2 z1 z2: term_pinterp I t1 z1 → term_pinterp I t2 z2 → fsatp I (f_lt t1
t2) (if Z_lt_dec z1 z2 then true else false)
| fsatp_f_and f1 f2 b1 b2: fsatp I f1 b1 → fsatp I f2 b2 → fsatp I (f_and f1 f2) (b1 &&
b2)
| fsatp_f_not f b: fsatp I f b → fsatp I (f_not f) (negb b)
.
```

18.2 Order on partial interpretations

```
Inductive option_le {A}: option A → option A → Prop :=
```

option_le_None o: **option_le** None o
| option_le_Some v: **option_le** (Some v) (Some v).

Lemma option_le_refl A (o: **option** A): **option_le** o o.

Lemma option_le_trans A ($o1$ $o2$ $o3$: **option** A):
option_le $o1$ $o2$ → **option_le** $o2$ $o3$ → **option_le** $o1$ $o3$.

Definition pinterp_le (I I': pinterp) := ∀ s, **option_le** (I s) (I' s).

Lemma pinterp_le_refl I: pinterp_le I I.

Lemma pinterp_le_trans I I' I'': pinterp_le I I' → pinterp_le I' I'' → pinterp_le I I''.

18.3 Facts

Lemma term_pinterp_le I I' t v: pinterp_le I I' → **term_pinterp** I t v → **term_pinterp** I' t v.

Lemma fsatp_pinterp_le I I' f b: pinterp_le I I' → **fsatp** I f b → **fsatp** I' f b.

Definition pinterp_update (I: pinterp) x v := fun y ⇒ if eq_nat_dec x y then Some v else I y.

Lemma pinterp_le_update I s v: I s = None → pinterp_le I (pinterp_update I s v).

Definition dom (I: pinterp) := fun s ⇒ I s ≠ None.
Definition val (I: pinterp) := fun s ⇒ match I s with None ⇒ $0\%Z$ | Some z ⇒ z end.

Lemma dom_pinterp_update I s v: dom (pinterp_update I s v) = **Union** _ (dom I) (**Singleton** _ s).

Lemma term_pinterp_interp I t v: **term_pinterp** I t v → term_interp (val I) t = v.

Lemma fsatp_fsat I f: ∀ b, **fsatp** I f b → if b then fsat (val I) f else ¬ fsat (val I) f.

Lemma term_pinterp_symbols I t v:
term_pinterp I t v → elems (term_symbols t) <_ dom I.

Lemma term_symbols_term_pinterp I t:
elems (term_symbols t) <_ dom I → ∃ v, **term_pinterp** I t v.

Lemma fsatp_formula_symbols I f: ∀ b, **fsatp** I f b → elems (formula_symbols f) <_ dom I.

Lemma formula_symbols_fsatp I f:
elems (formula_symbols f) <_ dom I → ∃ b, **fsatp** I f b.

Chapter 19

Library SymbolicSoundness

```
Require Export SemiconcreteExecutionFacts.
Require Export SymbolicExecutionFacts.
Require Export PartialInterpretations.
Require Export List.
```

19.1 Interpretation of symbolic states

```
Definition store_interp I (s: sstore): store := fun x ⇒ term_interp I (sstore_lookup s x).
Definition chunk_interp I (sc: schunk): chunk :=
  let (p, vs) := sc in
  Chunk p (map (term_interp I) vs).
Definition heap_of_chunks(cs: list chunk): heap :=
  fold_right (fun c h ⇒ heap_add (singleton_heap c) h) empty_heap cs.
Definition heap_interp I (sh: sheap): heap := heap_of_chunks (map (chunk_interp I) sh).

Definition store_pinterp I (ss: sstore) (s: store): Prop :=
  ∀ x, term_pinterp I (sstore_lookup ss x) (s x).
Inductive chunk_pinterp I: schunk → chunk → Prop :=
  chunk_pinterp_intro p ts vs:
  Forall2 (term_pinterp I) ts vs → chunk_pinterp I (SChunk p ts) (Chunk p vs)
.
Inductive heap_pinterp I (sh: sheap): heap → Prop :=
  heap_pinterp_intro cs: Forall2 (chunk_pinterp I) sh cs → heap_pinterp I sh (heap_of_chunks
cs).

Lemma heap_of_chunks_cons c cs: heap_of_chunks (c::cs) = heap_add (singleton_heap c)
(heap_of_chunks cs).

Lemma heap_of_chunks_app cs1 cs2:
heap_of_chunks (cs1 ++ cs2) = heap_add (heap_of_chunks cs1) (heap_of_chunks cs2).
```

Lemma store_pinterp_le I I' ss s: pinterp_le I I' $\rightarrow$ store_pinterp I ss s $\rightarrow$ store_pinterp I' ss s.

Lemma chunk_pinterp_le I I' sc c: pinterp_le I I' $\rightarrow$ **chunk_pinterp** I sc c $\rightarrow$ **chunk_pinterp** I' sc c.

Lemma heap_pinterp_le I I' sh h: pinterp_le I I' $\rightarrow$ **heap_pinterp** I sh h $\rightarrow$ **heap_pinterp** I' sh h.

Lemma store_pinterp_sstore0 I: store_pinterp I sstore0 store0.

Lemma store_pinterp_sstore_update I ss x t s v:
store_pinterp I ss s $\rightarrow$
term_pinterp I t v $\rightarrow$
store_pinterp I (sstore_update ss x t) (store_update s x v).

Lemma store_pinterp_sstore_updates I ss s xs ts vs:
store_pinterp I ss s $\rightarrow$
Forall2 (**term_pinterp** I) ts vs $\rightarrow$
store_pinterp I (sstore_updates ss xs ts) (store_updates s xs vs).

19.2 The representation relation

```
Inductive represents(I: pinterp): sstate → state → Prop :=
  represents_intro pc ss sh s h:
  dom I = elems (formula_symbols pc) →
  fsatp I pc true →
  store_pinterp I ss s →
  heap_pinterp I sh h →
  represents I (SState pc ss sh) (s, h).
```

```
Definition rhol(I: pinterp): sstate → scoutcome pinterp :=
  fun st ⇒
  demonicT (fun I' ⇒
  demonicT (fun st' ⇒
  demonicT (fun _: pinterp_le I I' ∧ represents I' st st' ⇒
  single st' I'
  ))).
```

19.3 Soundness of the symbolic mutators with respect to their semiconcrete counterparts

```
Definition approxl {A} (I: pinterp) (C: smutatora A) (C': scmutatora A): Prop :=
  C;, rhol I ==> rhol I; C'.
```

```
Notation "C ~~>[ I ] C'" := (approxI I C C') (at level 55).

Definition approx {A} (C: smutatora A) (C': scmutatora A): Prop :=
  ∀ I, C ~~>[I] C'.

Infix "~~>" := approx (at level 55).

Section AnnotatedProgram.

Lemma fresh_None I f: dom I = elems (formula_symbols f) → I (fresh_symbol (formula_symbols
f)) = None.

Lemma represents_fresh I pc ss sh st' v:
  let sym := fresh_symbol (formula_symbols pc) in
  let t := t_symb sym in
  represents I (SState pc ss sh) st' →
  represents
    (pinterp_update I sym v)
    (SState (f_and pc (f_eq t t)) ss sh)
    st'.

Lemma fresh_mut_approx A I (f1: term → smutatora A) (f2: Z → scmutatora A):
  (∀ t I' v, pinterp_le I I' → term_pinterp I' t v → f1 t ~~>[I'] f2 v) →
  seqf fresh_mut f1 ~~>[I] seqf (pick_demonic Z) f2.

Lemma seq_approx I A1 (C1: smutatora A1) A2 (C2: smutatora A2) (C1': scmutatora A1)
(C2': scmutatora A2):
  C1 ~~>[I] C1' → C2 ~~>[I] C2' → C1; C2 ~~>[I] C1';C2'.

Lemma seq_approx' I A1 (C1: smutatora A1) A2 (C2: smutatora A2) A1' (C1': scmutatora
A1') (C2': scmutatora A2):
  C1; noop ~~>[I] C1'; noop → C2 ~~>[I] C2' → C1; C2 ~~>[I] C1'; C2'.

Lemma covers_approx I A (C1 C2: smutatora A) C':
  C1 ==> C2 → C2 ~~>[I] C' → C1 ~~>[I] C'.

Lemma approx_covers I A C (C'1 C'2: scmutatora A):
  C'1 ==> C'2 → C ~~>[I] C'1 → C ~~>[I] C'2.

Lemma set_sstore_approx I ss s A (f: sstore → smutatora A) f':
  store_pinterp I ss s →
  (∀ I' ss0 s0, pinterp_le I I' → store_pinterp I' ss0 s0 → f ss0 ~~>[I'] f' s0) →
  seqf (set_sstore ss) f ~~>[I] seqf (set_store s) f'.

Lemma seqf_approx I A B (C: smutatora A) (f: A → smutatora B) C' f':
  C ~~>[I] C' → (∀ x, f x ~~>[I] f' x) → seqf C f ~~>[I] seqf C' f'.

Lemma seqf_seq_approx I (C1: smutator) A (C2: smutatora A) B (f: A → smutatora B) C1'
A' (C2': scmutatora A') f':
  C1 ~~>[I] C1' →
  seqf C2 f ~~>[I] seqf C2' f' →
  seqf (C1; C2) f ~~>[I] seqf (C1'; C2') f'.
```

```
Lemma swith_store_approx I ss A (C: smutatora A) s C':
  store_pinterp I ss s →
  (∀ I', pinterp_le I I' → C ~~>[I'] C') →
  swith_store ss C ~~>[I] with_store s C'.

Lemma swith_store_approx' I ss (C: smutatora term) A (f: term → smutatora A) s C' f':
  store_pinterp I ss s →
  (∀ I' A (g: term → smutatora A) g',
   (∀ I'' t v, pinterp_le I' I'' → term_pinterp I'' t v → g t ~~>[I''] g' v) →
   pinterp_le I I' → seqf C g ~~>[I'] seqf C' g') →
  (∀ I' t v, pinterp_le I I' → term_pinterp I' t v → f t ~~>[I'] f' v) →
  seqf (swith_store ss C) f ~~>[I] seqf (with_store s C') f'.

Lemma swith_store'_approx I ss (C: smutator) A (f: sstore → smutatora A) s C' f':
  store_pinterp I ss s →
  (∀ I', pinterp_le I I' → C ~~>[I'] C') →
  (∀ I' ss0 s0, pinterp_le I I' → store_pinterp I' ss0 s0 → f ss0 ~~>[I'] f' s0) →
  seqf (swith_store' ss C) f ~~>[I] seqf (with_store' s C') f'.

Lemma get_store_approx I {A} (f: sstore → smutatora A) f':
  (∀ I' ss s, pinterp_le I I' → store_pinterp I' ss s → f ss ~~>[I'] f' s) →
  seqf get_sstore f ~~>[I] seqf get_store f'.

Lemma yield_approx I {A} (v: A): yield v ~~>[I] yield v.

Lemma swith_local_store_approx I C C':
  (∀ I', pinterp_le I I' → C ~~>[I'] C') →
  swith_local_store C ~~>[I] with_local_store C'.

Lemma message_mut_approx I msg A (C: smutatora A) (C': scmutatora A):
  C ~~>[I] C' →
  message_mut msg; C ~~>[I] C'.

Lemma sclear_heap_approx I: sclear_heap ~~>[I] clear_heap.

Lemma sleakcheck_approx I: sleakcheck ~~>[I] leakcheck.

Variable rspecs: spec_table.
Variable pdefs: pred_table.
Variable rdefs: routine_table.

Lemma seval_eval I ss s e: store_pinterp I ss s → term_pinterp I (seval ss e) (eval s e).

Lemma sevals_evals I ss s es:
  store_pinterp I ss s →
  Forall2 (term_pinterp I)
    (map (seval ss) es)
    (map (eval s) es).

Lemma seval_eval' I ss s e: store_pinterp I ss s → term_interp (val I) (seval ss e) = eval s e.
```

Lemma sbeval_beval I ss s b: store_pinterp I ss s $\rightarrow$ (fsat (val I) (sbeval ss b) $\leftrightarrow$ beval s b = true).

Lemma sassert_bexpr_approx I b: sassert_bexpr b ~~>[I] assert_bexpr b.

Lemma seval_mut_approx I e A (f: **term** $\rightarrow$ smutatora A) (f': **Z** $\rightarrow$ scmutatora A):
  ($\forall$ I' t v, pinterp_le I I' $\rightarrow$ **term_pinterp** I' t v $\rightarrow$ f t ~~>[I'] f' v) $\rightarrow$
  seqf (seval_mut e) f ~~>[I] seqf (eval_mut e) f'.

Lemma sevals_mut_approx I es A (f: **list term** $\rightarrow$ smutatora A) (f': **list Z** $\rightarrow$ scmutatora A):
  ($\forall$ I' ts vs, pinterp_le I I' $\rightarrow$ **Forall2** (**term_pinterp** I') ts vs $\rightarrow$ f ts ~~>[I'] f' vs) $\rightarrow$
  seqf (sevals_mut es) f ~~>[I] seqf (evals_mut es) f'.

Lemma forall2b_follows_eq $ts1$ $ts2$ I pc:
  fsat I pc $\rightarrow$
  length $ts1$ = length $ts2$ $\rightarrow$
  forall2b $ts1$ $ts2$ (`fun` $t1$ $t2$ $\Rightarrow$ follows pc (f_eq $t1$ $t2$)) = true $\rightarrow$
  map (term_interp I) $ts1$ = map (term_interp I) $ts2$.

Lemma Forall2_term_pinterp_interp I ts vs:
  Forall2 (**term_pinterp** I) ts vs $\rightarrow$
  map (term_interp (val I)) ts = vs.

Lemma map_firstn A B (xs: **list** A) (f: $A \rightarrow B$) n:
  map f (firstn n xs) = firstn n (map f xs).

Lemma Forall2_skipn A B (P: $A \rightarrow B \rightarrow$ `Prop`) xs ys n:
  Forall2 P xs ys $\rightarrow$ **Forall2** P (skipn n xs) (skipn n ys).

Lemma extract0_match_chunk:
  $\forall$ $todo$ I h pc p $ts1$ n $done$ rcs $ts2$,
  extract0 (match_chunk pc p $ts1$ n) $todo$ $done$ = Some (rcs, $ts2$) $\rightarrow$
  heap_pinterp I ($done$ ++ $todo$) h $\rightarrow$
  fsatp I pc true $\rightarrow$
  length $ts2$ = n $\wedge$
  $\exists$ $vs1$ $vs2$ h',
  map (term_interp (val I)) $ts1$ = $vs1$ $\wedge$
  Forall2 (**term_pinterp** I) $ts2$ $vs2$ $\wedge$
  h = heap_add (singleton_heap (Chunk p ($vs1$ ++ $vs2$))) h' $\wedge$
  heap_pinterp I rcs h'.

Lemma extract_match_chunk I h pc p $ts1$ n sh rcs $ts2$:
  extract (match_chunk pc p $ts1$ n) sh = Some (rcs, $ts2$) $\rightarrow$
  heap_pinterp I sh h $\rightarrow$
  fsatp I pc true $\rightarrow$
  length $ts2$ = n $\wedge$
  $\exists$ $vs1$ $vs2$ h',
  map (term_interp (val I)) $ts1$ = $vs1$ $\wedge$
  Forall2 (**term_pinterp** I) $ts2$ $vs2$ $\wedge$

h = heap_add (singleton_heap (Chunk p ($vs1$ ++ $vs2$))) h' ∧
heap_pinterp I rcs h'.

Lemma Forall2_term_pinterp_le I I' ts vs:
pinterp_le I I' →
Forall2 (**term_pinterp** I) ts vs →
Forall2 (**term_pinterp** I') ts vs.

Lemma Forall2_length A B xs ys (P: A → B → `Prop`):
Forall2 P xs ys →
length ys = length xs.

Lemma extract_chunk_approx I p ts vs n A (f: **list term** → smutatora A) (f': **list Z** → scmutatora A):
Forall2 (**term_pinterp** I) ts vs →
(∀ I' ts' vs', pinterp_le I I' → length ts' = n → **Forall2** (**term_pinterp** I') ts' vs' → f ts' `~~>`[I'] f' vs') →
seqf (extract_chunk p ts n) f `~~>`[I] seqf (cons_chunk p vs n) f'.

Lemma extract_chunk_1_1_approx I p t v A (f: **term** → smutatora A) (f': **Z** → scmutatora A):
term_pinterp I t v →
(∀ I' t' v', pinterp_le I I' → **term_pinterp** I' t' v' → f t' `~~>`[I'] f' v') →
seqf (extract_chunk_1_1 p t) f `~~>`[I] seqf (cons_chunk_1_1 p v) f'.

Lemma update_sstore_approx I x t v:
term_pinterp I t v →
update_sstore x t `~~>`[I] update_store x v.

Lemma update_sstore_n_approx I xs ts vs:
Forall2 (**term_pinterp** I) ts vs →
update_sstore_n xs ts `~~>`[I] update_store_n xs vs.

Lemma havoc_approx I xs: shavoc xs `~~>`[I] havoc xs.

Lemma fork_approx I A ($C1$ $C2$: smutatora A) ($C1'$ $C2'$: scmutatora A):
$C1$ `~~>`[I] $C1'$ →
$C2$ `~~>`[I] $C2'$ →
fork $C1$ $C2$ `~~>`[I] fork $C1'$ $C2'$.

Lemma term_symbols_seval I ss s e:
store_pinterp I ss s → elems (term_symbols (seval ss e)) <_ dom I.

Lemma formula_symbols_sbeval I ss s b:
store_pinterp I ss s → elems (formula_symbols (sbeval ss b)) <_ dom I.

Lemma sassume_bexpr_approx I b: sassume_bexpr b `~~>`[I] assume_bexpr b.

Lemma sprod_chunk_approx I p ts vs:
Forall2 (**term_pinterp** I) ts vs →
sprod_chunk (SChunk p ts) `~~>`[I] prod_chunk (Chunk p vs).

Lemma assume_formula_approx I f (P: Prop):
($P \to$ **fsatp** I f true) $\to$
assume_formula f ~~>[I] assume_prop P.

Lemma extract_pointsto_approx I tl l A (f: **term** $\to$ smutatora A) (f': **Z** $\to$ scmutatora A):
term_pinterp I tl l $\to$
($\forall$ I' tv v, pinterp_le I I' $\to$ **term_pinterp** I' tv v $\to$ f tv ~~>[I'] f' v) $\to$
seqf (extract_pointsto tl) f ~~>[I] seqf (cons_pointsto l) f'.

Lemma extract_malloc_block_approx I tl l A (f: **term** $\to$ smutatora A) (f': **Z** $\to$ scmutatora A):
term_pinterp I tl l $\to$
($\forall$ I' tv v, pinterp_le I I' $\to$ **term_pinterp** I' tv v $\to$ f tv ~~>[I'] f' v) $\to$
seqf (extract_malloc_block tl) f ~~>[I] seqf (cons_malloc_block l) f'.

Lemma fail_approx I A (C: scmutatora A): fail ~~>[I] C.

Lemma message_mut_approx_noop I msg: message_mut msg ~~>[I] noop.

Lemma assume_formula_lt_0 I t v:
term_pinterp I t v $\to$
assume_formula (f_lt (t_lit 0) t) ~~>[I] assume_prop (0 < v)%Z.

Lemma term_pinterp_add_1 I t v:
term_pinterp I t v $\to$
term_pinterp I (t_add t (t_lit 1)) (v + 1)%Z.

Lemma term_pinterp_lit I z: **term_pinterp** I (t_lit z) z.

Lemma sprod_chunk_spointsto_approx I tl tv l v:
term_pinterp I tl l $\to$
term_pinterp I tv v $\to$
sprod_chunk (spointsto tl tv) ~~>[I] prod_chunk (pointsto l v).

Lemma sprod_chunk_smalloc_block_approx I tl tv l v:
term_pinterp I tl l $\to$
term_pinterp I tv v $\to$
sprod_chunk (smalloc_block tl tv) ~~>[I] prod_chunk (malloc_block l v).

Lemma iter_fresh_approx n I (f: _ $\to$ smutator) (f': _ $\to$ scmutator):
($\forall$ I' ts vs, pinterp_le I I' $\to$ **Forall2** (**term_pinterp** I') ts vs $\to$ f ts ~~>[I'] f' vs) $\to$
seqf (iter fresh_mut n) f ~~>[I] seqf (iter (pick_demonic **Z**) n) f'.

Hint Unfold approx : *approx*.

Hint Resolve
Forall2_app
assume_formula_lt_0
term_pinterp_lit
term_pinterp_add_1
sprod_chunk_spointsto_approx

sprod_chunk_smalloc_block_approx
fresh_mut_approx
iter_fresh_approx
sassert_bexpr_approx
message_mut_approx
seval_mut_approx
sevals_mut_approx
extract_chunk_approx
extract_chunk_1_1_approx
update_sstore_approx
update_sstore_n_approx
havoc_approx
fork_approx
seq_approx
sassume_bexpr_approx
message_mut_approx
term_pinterp_le
Forall2_term_pinterp_le
sprod_chunk_approx
extract_pointsto_approx
extract_malloc_block_approx
swith_store_approx
swith_store_approx'
swith_store'_approx
swith_local_store_approx
store_pinterp_sstore_update
store_pinterp_sstore_updates
store_pinterp_sstore0
store_pinterp_le
fail_approx
message_mut_approx_noop
sclear_heap_approx
sleakcheck_approx
: *approx*.

Lemma sprod_chunks_approx I tl l n:
term_pinterp I tl l $\rightarrow$ sprod_pointstos tl n ~~>[I] prod_pointstos l n.

Lemma extract_pointstos_approx I tl l n:
term_pinterp I tl l $\rightarrow$
extract_pointstos tl n ~~>[I] cons_pointstos l n.

Lemma constant_term_value_approx I tn n:
term_pinterp I tn n $\rightarrow$
constant_term_value tn ~~>[I] yield n.

Lemma extract_pointstos_approx' I tl l tn n:
  term_pinterp I tl l $\rightarrow$
  term_pinterp I tn n $\rightarrow$
  $n \leftarrow$ constant_term_value tn; extract_pointstos tl (Z.to_nat n) ~~>[I] cons_pointstos l (Z.to_nat n).

Lemma sconsume_approx I a: sconsume a ~~>[I] consume a.

Lemma sproduce_approx I a: sproduce a ~~>[I] produce a.

Hint Resolve sprod_chunks_approx extract_pointstos_approx' sconsume_approx sproduce_approx : *approx*.

Definition s_exec := s_exec *rspecs* *pdefs*.
Definition sc_exec := sc_exec *rspecs* *pdefs*.

Lemma s_exec_approx I c: s_exec c ~~>[I] sc_exec c.

Lemma eresult_bind_ok $\{A\}$ $\{r1\ r2$: **eresult** $A\}$:
  eresult_bind $r1$ $r2$ = ok $\rightarrow$ $r1$ = ok $\land$ $r2$ = ok.

Lemma foralle_Forall $\{A\ B\}$ (f: $A \rightarrow$ **eresult** B) xs:
  foralle f xs = ok $\rightarrow$ **Forall** (fun $x \Rightarrow f\ x$ = ok) xs.

Lemma eresult_annotate_ok msg r:
  eresult_annotate msg r = ok $\rightarrow$ r = ok.

Lemma sresult0_ok_safe A (r: soutcome A):
  $\forall$ msg, sresult_ok0 msg r = ok $\rightarrow$ safe r.

Lemma sresult_ok_safe A (r: soutcome A):
  sresult_ok r = ok $\rightarrow$ safe r.

Definition I0: pinterp := fun _ $\Rightarrow$ None.

Lemma represents_sstate0: **represents** I0 sstate0 state0.

Lemma approx_safe (C: smutator) (C': scmutator): C ~~> C' $\rightarrow$ safe (sstate0 |> C) $\rightarrow$ safe (state0 |> C').

Lemma svalid_routine_sound r $rspec$:
  svalid_routine *rspecs* *pdefs* *rdefs* r *rspec* = ok $\rightarrow$
  valid_routine *rspecs* *pdefs* *rdefs* r *rspec*.

Lemma svalid_command_sound c:
  svalid_command *rspecs* *pdefs* c = ok $\rightarrow$
  valid_command *rspecs* *pdefs* c.

Lemma svalid_routines_sound:
  svalid_routines *rspecs* *pdefs* *rdefs* = ok $\rightarrow$
  valid_routines *rspecs* *pdefs* *rdefs*.

Theorem symbolic_soundness c:
  svalid_program *rspecs* *pdefs* *rdefs* c = ok $\rightarrow$

valid_program *rspecs pdefs rdefs c*.

End AnnotatedProgram.

Chapter 20

Library Soundness

```
Require Export SemiconcreteSoundness.
Require Export SymbolicSoundness.
```

```
Theorem soundness rspecs pdefs rdefs c:
  svalid_program rspecs pdefs rdefs c = ok →
  cvalid_program rdefs c.
```

```
Goal svalid_program my_rspecs my_preddefs my_rdefs reverse_test = ok.
```

```
Goal ¬ cvalid_program my_rt reverse_test_broken.
```

```
Goal cvalid_program my_rdefs reverse_test.
```

```
Print Assumptions soundness.
```