状态关系单子与霍尔逻辑

1 状态关系单子程序

下面用 StateRelMonad 表示带有程序状态的非确定性计算。

```
Module StateRelMonad.
```

```
Definition M (Σ A: Type): Type :=
  Σ -> A -> Σ -> Prop.
```

```
Definition bind (Σ A B: Type) (f: M Σ A) (g: A -> M Σ B): M Σ B :=
  fun (s1: Σ) (b: B) (s3: Σ) =>
    exists (a: A) (s2: Σ),
      (s1, a, s2) ∈ f /\ (s2, b, s3) ∈ g a.
```

```
Definition ret (Σ A: Type) (a0: A): M Σ A :=
  fun (s1: Σ) (a: A) (s2: Σ) => a = a0 /\ s1 = s2.
```

```
End StateRelMonad.
```

```
#[export] Instance state_rel_monad (Σ: Type): Monad (StateRelMonad.M Σ) :=
{|
  bind := StateRelMonad.bind Σ;
  ret := StateRelMonad.ret Σ;
|}.
```

以下可以再定义一些额外的算子。

```
Definition assume {Σ: Type} (P: Σ -> Prop): StateRelMonad.M Σ unit :=
  fun s1 _ s2 => P s1 /\ s1 = s2.
```

```
Definition choice {Σ A: Type} (f g: StateRelMonad.M Σ A): StateRelMonad.M Σ A :=
  f ∪ g.
```

```
Definition any {Σ: Type} (A: Type): StateRelMonad.M Σ A :=
  fun s1 _ s2 => s1 = s2.
```

```
Definition repeat_break_f
           {Σ A B: Type}
           (body: A -> StateRelMonad.M Σ (ContinueOrBreak A B))
           (W: A -> StateRelMonad.M Σ B)
           (a: A): StateRelMonad.M Σ B :=
  x <- body a;;
  match x with
  | by_continue a' => W a'
  | by_break b => ret b
  end.
```

```
Definition repeat_break
           {Σ A B: Type}
           (body: A -> StateRelMonad.M Σ (ContinueOrBreak A B)):
  A -> StateRelMonad.M Σ B :=
  Kleene_LFix (repeat_break_f body).
```

```
Definition continue {Σ A B: Type} (a: A):
  StateRelMonad.M Σ (ContinueOrBreak A B) :=
  ret (by_continue a).
```

```
Definition break {Σ A B: Type} (b: B):
  StateRelMonad.M Σ (ContinueOrBreak A B) :=
  ret (by_break b).
```

2 霍尔逻辑

在 StateRelMonad 上的霍尔逻辑是一个关于霍尔三元组的逻辑。

```
Definition Hoare {Σ A: Type}
                 (P: Σ -> Prop)
                 (c: StateRelMonad.M Σ A)
                 (Q: A -> Σ -> Prop): Prop :=
  forall s1 a s2, P s1 -> (s1, a, s2) ∈ c -> Q a s2.
```

```
Theorem Hoare_bind {Σ A B: Type}:
  forall (P: Σ -> Prop)
         (f: StateRelMonad.M Σ A)
         (Q: A -> Σ -> Prop)
         (g: A -> StateRelMonad.M Σ B)
         (R: B -> Σ -> Prop),
  Hoare P f Q ->
  (forall a, Hoare (Q a) (g a) R) ->
  Hoare P (bind f g) R.
```

```
Theorem Hoare_ret {Σ A: Type}:
  forall (P: A -> Σ -> Prop) (a0: A),
    Hoare (P a0) (ret a0) P.
```

```
Theorem Hoare_choice {Σ A: Type}:
  forall P (f g: StateRelMonad.M Σ A) Q,
    Hoare P f Q ->
    Hoare P g Q ->
    Hoare P (choice f g) Q.
```

```
Theorem Hoare_assume_bind {Σ A: Type}:
  forall P (Q: Σ -> Prop) (f: StateRelMonad.M Σ A) R,
    Hoare (fun s => Q s /\ P s) f R ->
    Hoare P (assume Q;; f) R.
```

```
Theorem Hoare_any_bind {Σ A B: Type}:
  forall P Q (f: A -> StateRelMonad.M Σ B),
    (forall a, Hoare P (f a) Q) ->
    (Hoare P (bind (any A) f) Q).
```

```
Theorem Hoare_conseq {Σ A: Type}:
  forall (P1 P2: Σ -> Prop) f (Q1 Q2: A -> Σ -> Prop),
    (forall s, P1 s -> P2 s) ->
    (forall b s, Q2 b s -> Q1 b s) ->
    Hoare P2 f Q2 ->
    Hoare P1 f Q1.
```

```
Theorem Hoare_conseq_pre {Σ A: Type}:
  forall (P1 P2: Σ -> Prop) f (Q: A -> Σ -> Prop),
    (forall s, P1 s -> P2 s) ->
    Hoare P2 f Q ->
    Hoare P1 f Q.
```

```
Theorem Hoare_conseq_post {Σ A: Type}:
  forall (P: Σ -> Prop) f (Q1 Q2: A -> Σ -> Prop),
    (forall b s, Q2 b s -> Q1 b s) ->
    Hoare P f Q2 ->
    Hoare P f Q1.
```

```
Theorem Hoare_conj {Σ A: Type}:
  forall (P: Σ -> Prop) f (Q1 Q2: A -> Σ -> Prop),
    Hoare P f Q1 ->
    Hoare P f Q2 ->
    Hoare P f (fun a s => Q1 a s /\ Q2 a s).
```

```
Theorem Hoare_forall {Σ A: Type}:
  forall (X: Type) (P: Σ -> Prop) f (Q: X -> A -> Σ -> Prop),
    (forall x, Hoare P f (Q x)) ->
    Hoare P f (fun a s => forall x, Q x a s).
```

```
Theorem Hoare_pre_ex {Σ A: Type}:
  forall (X: Type) (P: X -> Σ -> Prop) f (Q: A -> Σ -> Prop),
    (forall x, Hoare (P x) f Q) ->
    Hoare (fun s => exists x, P x s) f Q.
```

```
Theorem Hoare_ret' {Σ A: Type}:
  forall (P: Σ -> Prop) (Q: A -> Σ -> Prop) (a0: A),
    (forall s, P s -> Q a0 s) ->
    Hoare P (ret a0) Q.
```

```
Theorem Hoare_repeat_break {Σ A B: Type}:
  forall (body: A -> StateRelMonad.M Σ (ContinueOrBreak A B))
         (P: A -> Σ -> Prop)
         (Q: B -> Σ -> Prop),
    (forall a, Hoare (P a) (body a) (fun x s => match x with
                                                | by_continue a => P a s
                                                | by_break b => Q b s
                                                end)) ->
    (forall a, Hoare (P a) (repeat_break body a) Q).
```

3 定义有向图和图上的程序

可以如下定义有向图。

```
Record PreGraph (Vertex Edge: Type) := {
  vvalid : Vertex -> Prop;
  evalid : Edge -> Prop;
  src : Edge -> Vertex;
  dst : Edge -> Vertex
}.
```

基于此就能定义 “从 x 点经过一条边可以到达 y 点”。

```
Record step_aux {V E: Type} (pg: PreGraph V E) (e: E) (x y: V): Prop :=
{
  step_evalid: pg.(evalid) e;
  step_src_valid: pg.(vvalid) x;
  step_dst_valid: pg.(vvalid) y;
  step_src: pg.(src) e = x;
  step_dst: pg.(dst) e = y;
}.
```

```
Definition step {V E: Type} (pg: PreGraph V E) (x y: V): Prop :=
  exists e, step_aux pg e x y.
```

进一步，单步可达关系的自反传递闭包就是多步可达关系。

```
Definition reachable {V E: Type} (pg: PreGraph V E) :=
  clos_refl_trans (step pg).
```

自反传递闭包 `clos_refl_trans` 是 SetsClass 库提供的定义，该库还提供了相应的证明方式。

- `reflexivity` ：利用其自反性证明；
- `transitivity` ：利用其传递性证明；
- `transitivity_1n` ：利用其传递性证明并专门支持前一步是单步的情况；
- `transitivity_1n` ：利用其传递性证明并专门支持后一步是单步的情况；

- `induction_1n` ：从前往后对步数归纳；
- `induction_n1` ：从后往前对步数归纳；