软件理论基础与实践

EQUIV: Program Equivalence

胡振江 熊英飞
北京大学

行为等价

- 之前定义了无状态情况下的表达式等价
- 如何定义带状态的IMP程序等价性?

行为等价

对于函数和关系采取不同的定义

```
Definition aequiv (a1 a2 : aexp) : Prop :=
  forall (st : state),
    aeval st a1 = aeval st a2.

Definition bequiv (b1 b2 : bexp) : Prop :=
  forall (st : state),
    beval st b1 = beval st b2.

Definition cequiv (c1 c2 : com) : Prop :=
  forall (st st' : state),
    (st =[ c1 ]=> st') <-> (st =[ c2 ]=> st').
```

等价程序示例

```
Theorem aequiv_example: aequiv <{ X - X }> <{ 0 }>.
Proof.
  intros st. simpl. lia.
Qed.

Theorem bequiv_example: bequiv <{ X - X = 0 }> <{ true }>.
Proof.
  intros st. unfold beval.
  rewrite aequiv_example. reflexivity.
Qed.
```

等价程序示例

```
Theorem skip_left : forall c,
  cequiv
    <{ skip; c }>
    c.
Proof.
  intros c st st'.
(** [Coq Proof View]
 * 1 subgoal
 *
 *   c : com
 *   st, st' : state
 *   ========================================
 *   st =[ skip; c ]=> st' <-> st =[ c ]=> st'
 *)
```

等价程序示例

```
 split; intros H.
(** [Coq Proof View]
 * 2 subgoals
 *
 *   c : com
 *   st, st' : state
 *   H : st =[ skip; c ]=> st'
 *   =========================
 *   st =[ c ]=> st'
 *
 * subgoal 2 is:
 *  st =[ skip; c ]=> st'
 *)
```

等价程序示例

```
- inversion H. subst.
(** [Coq Proof View]
 * 1 subgoal
 *
 *   c : com
 *   st, st' : state
 *   H : st =[ skip; c ]=> st'
 *   st'0 : state
 *   H2 : st =[ skip ]=> st'0
 *   H5 : st'0 =[ c ]=> st'
 *   =========================
 *   st =[ c ]=> st'
 *)
```

等价程序示例

```
    inversion H2. subst.
(** [Coq Proof View]
 * 1 subgoal
 *
 *   c : com
 *   st', st'0 : state
 *   H2 : st'0 =[ skip ]=> st'0
 *   H : st'0 =[ skip; c ]=> st'
 *   H5 : st'0 =[ c ]=> st'
 *   ===========================
 *   st'0 =[ c ]=> st'
 *)
    assumption.
```

等价程序示例

```
- (* <- *)
    (** [Coq Proof View]
 * 1 subgoal
 *
 *   c : com
 *   st, st' : state
 *   H : st =[ c ]=> st'
 *   =====================
 *   st =[ skip; c ]=> st'
 *)
```

等价程序示例

```
  apply E_Seq with st.
(** [Coq Proof View]
 * 2 subgoals
 *
 *   c : com
 *   st, st' : state
 *   H : st =[ c ]=> st'
 *   ===================
 *   st =[ skip ]=> st
 *
 * subgoal 2 is:
 *  st =[ c ]=> st'
 *)
  apply E_Skip. assumption.
Qed.
```

等价程序示例

```
Theorem if_true: forall b c1 c2,
  bequiv b <{true}>  ->
  cequiv <{ if b then c1 else c2 end }> c1.
Proof.
  intros b c1 c2 Hb.
  split; intros H.
  - (* -> *)
    inversion H; subst.
    + (* b evaluates to true *)
      assumption.
    + (* b evaluates to false (contradiction) *)
      unfold bequiv in Hb. simpl in Hb.
      rewrite Hb in H5.
      discriminate.
  - (* <- *)
    apply E_IfTrue; try assumption.
    unfold bequiv in Hb. simpl in Hb.
    apply Hb. Qed.
```

行为等价是等价关系

```
Lemma refl_aequiv : forall (a : aexp), aequiv a a.
Proof.
  intros a st. reflexivity.  Qed.

Lemma sym_aequiv : forall (a1 a2 : aexp),
  aequiv a1 a2 -> aequiv a2 a1.
Proof.
  intros a1 a2 H. intros st. symmetry. apply H.  Qed.

Lemma trans_aequiv : forall (a1 a2 a3 : aexp),
  aequiv a1 a2 -> aequiv a2 a3 -> aequiv a1 a3.
Proof.
  unfold aequiv. intros a1 a2 a3 H12 H23 st.
  rewrite (H12 st). rewrite (H23 st). reflexivity.  Qed.
```

行为等价是等价关系

```
Lemma refl_bequiv : forall (b : bexp), bequiv b b.
Proof.
  unfold bequiv. intros b st. reflexivity.  Qed.

Lemma sym_bequiv : forall (b1 b2 : bexp),
  bequiv b1 b2 -> bequiv b2 b1.
Proof.
  unfold bequiv. intros b1 b2 H. intros st. symmetry. app
ly H.  Qed.

Lemma trans_bequiv : forall (b1 b2 b3 : bexp),
  bequiv b1 b2 -> bequiv b2 b3 -> bequiv b1 b3.
Proof.
  unfold bequiv. intros b1 b2 b3 H12 H23 st.
  rewrite (H12 st). rewrite (H23 st). reflexivity.  Qed.
```

行为等价是等价关系

```
Lemma refl_cequiv : forall (c : com), cequiv c c.
Proof.
  unfold cequiv. intros c st st'. reflexivity.  Qed.

Lemma sym_cequiv : forall (c1 c2 : com),
  cequiv c1 c2 -> cequiv c2 c1.
Proof.
  unfold cequiv. intros c1 c2 H st st'.
  rewrite H. reflexivity.
Qed.

Lemma trans_cequiv : forall (c1 c2 c3 : com),
  cequiv c1 c2 -> cequiv c2 c3 -> cequiv c1 c3.
Proof.
  unfold cequiv. intros c1 c2 c3 H12 H23 st st'.
  rewrite H12. apply H23.
Qed.
```

同余关系Congruence relation

- 抽象代数中的同余关系是一种等价关系，如果子部件都满足该关系，则父部件也满足。
- 给定二元关系R，给定构造函数$f: A \to B$，如果$R(a, a') \to R\big(f(a), f(a')\big)$，则$R$对于$f$是一个同余关系。
  - A可以为一个tuple
- 行为等价也是一个同余关系

$$\frac{\texttt{aequiv a a'}}{\texttt{cequiv (x := a) (x := a')}} \qquad \frac{\texttt{cequiv } c_1 \ c_1' \qquad \texttt{cequiv } c_2 \ c_2'}{\texttt{cequiv } (c_1;c_2) \ (c_1';c_2')}$$

While同余关系证明

```
Theorem CWhile_congruence : forall b b' c c',
  bequiv b b' -> cequiv c c' ->
  cequiv <{ while b do c end }> <{ while b' do c' end }>.
Proof.
    assert (A: forall (b b' : bexp) (c c' : com) (st st' : state),
            bequiv b b' -> cequiv c c' ->
            st =[ while b do c end ]=> st' ->
            st =[ while b' do c' end ]=> st’).
  { unfold bequiv,cequiv.
    intros b b' c c' st st' Hbe Hc1e Hce.
    remember <{ while b do c end }> as cwhile
      eqn:Heqcwhile.
    induction Hce; inversion Heqcwhile; subst。
```

While同余关系证明

```
    + (* E_WhileFalse *)
      apply E_WhileFalse. rewrite <- Hbe. apply H.
    + (* E_WhileTrue *)
      apply E_WhileTrue with (st' := st').
      * (* show loop runs *) rewrite <- Hbe. apply H.
      * (* body execution *)
        apply (Hc1e st st').  apply Hce1.
      * (* subsequent loop execution *)
        apply IHHce2. reflexivity. }

  intros. split.
  - apply A; assumption.
  - apply A.
    + apply sym_bequiv. assumption.
    + apply sym_cequiv. assumption.
Qed.
```

程序变换

- 程序变换是从程序到程序的映射

```
Definition atrans_sound (atrans : aexp -> aexp) : Prop :=
  forall (a : aexp),
    aequiv a (atrans a).

Definition btrans_sound (btrans : bexp -> bexp) : Prop :=
  forall (b : bexp),
    bequiv b (btrans b).

Definition ctrans_sound (ctrans : com -> com) : Prop :=
  forall (c : com),
    cequiv c (ctrans c).
```

程序变换举例：常量传播

```
Fixpoint fold_constants_aexp (a : aexp) : aexp :=
  match a with
  | ANum n          => ANum n
  | AId x           => AId x
  | <{ a1 + a2 }>   =>
    match (fold_constants_aexp a1,
           fold_constants_aexp a2)
    with
    | (ANum n1, ANum n2) => ANum (n1 + n2)
    | (a1', a2') => <{ a1' + a2' }>
    end
```

程序变换举例：常量传播

```
| <{ a1 - a2 }> =>
  match (fold_constants_aexp a1,
         fold_constants_aexp a2)
  with
  | (ANum n1, ANum n2) => ANum (n1 - n2)
  | (a1', a2') => <{ a1' - a2' }>
  end
| <{ a1 * a2 }>  =>
  match (fold_constants_aexp a1,
         fold_constants_aexp a2)
  with
  | (ANum n1, ANum n2) => ANum (n1 * n2)
  | (a1', a2') => <{ a1' * a2' }>
  end
end.
```

程序变换举例：常量传播

```
Fixpoint fold_constants_bexp (b : bexp) : bexp :=
  match b with
  | <{true}>          => <{true}>
  | <{false}>         => <{false}>
  | <{ a1 = a2 }>  =>
      match (fold_constants_aexp a1,
             fold_constants_aexp a2) with
      | (ANum n1, ANum n2) =>
          if n1 =? n2 then <{true}> else <{false}>
      | (a1', a2') =>
          <{ a1' = a2' }>
      end
```

程序变换举例：常量传播

```
| <{ a1 <= a2 }>  =>
    match (fold_constants_aexp a1,
           fold_constants_aexp a2) with
    | (ANum n1, ANum n2) =>
        if n1 <=? n2 then <{true}> else <{false}>
    | (a1', a2') =>
        <{ a1' <= a2' }>
    end
| <{ ~ b1 }>  =>
    match (fold_constants_bexp b1) with
    | <{true}> => <{false}>
    | <{false}> => <{true}>
    | b1' => <{ ~ b1' }>
    end
```

程序变换举例：常量传播

```
| <{ b1 && b2 }>  =>
    match (fold_constants_bexp b1,
           fold_constants_bexp b2) with
    | (<{true}>, <{true}>) => <{true}>
    | (<{true}>, <{false}>) => <{false}>
    | (<{false}>, <{true}>) => <{false}>
    | (<{false}>, <{false}>) => <{false}>
    | (b1', b2') => <{ b1' && b2' }>
    end
end.
```

程序变换举例：常量传播

```
Fixpoint fold_constants_com (c : com) : com :=
  match c with
  | <{ skip }> =>
      <{ skip }>
  | <{ x := a }> =>
      <{ x := (fold_constants_aexp a) }>
  | <{ c1 ; c2 }>  =>
      <{ fold_constants_com c1 ; fold_constants_com c2 }>
```

程序变换举例：常量传播

```
| <{ if b then c1 else c2 end }> =>
    match fold_constants_bexp b with
    | <{true}>  => fold_constants_com c1
    | <{false}> => fold_constants_com c2
    | b' => <{ if b' then fold_constants_com c1
                     else fold_constants_com c2 end}>
    end
| <{ while b do c1 end }> =>
    match fold_constants_bexp b with
    | <{true}> => <{ while true do skip end }>
    | <{false}> => <{ skip }>
    | b' => <{ while b' do (fold_constants_com c1) end }>
    end
end.
```

常量传播的正确性

```
Theorem fold_constants_aexp_sound :
  atrans_sound fold_constants_aexp.
Proof.
  unfold atrans_sound. intros a. unfold aequiv. intros st.
  induction a; simpl;
    (* ANum and AId follow immediately *)
    try reflexivity;
    (* APlus, AMinus, and AMult follow from the IH
       and the observation that
              aeval st (<{ a1 + a2 }>)
            = ANum ((aeval st a1) + (aeval st a2))
            = aeval st (ANum ((aeval st a1) + (aeval st a2)))
       (and similarly for AMinus/minus and AMult/mult) *)
    try (destruct (fold_constants_aexp a1);
         destruct (fold_constants_aexp a2);
         rewrite IHa1; rewrite IHa2; reflexivity). Qed.
```

bexp和com的正确性证明留作作业

内联变量

```
Fixpoint subst_aexp (x : string) (u : aexp) (a : aexp) : aexp :=
  match a with
  | ANum n        =>
      ANum n
  | AId x'        =>
      if eqb_string x x' then u else AId x'
  | <{ a1 + a2 }>  =>
      <{ (subst_aexp x u a1) + (subst_aexp x u a2) }>
  | <{ a1 - a2 }> =>
      <{ (subst_aexp x u a1) - (subst_aexp x u a2) }>
  | <{ a1 * a2 }>  =>
      <{ (subst_aexp x u a1) * (subst_aexp x u a2) }>
  end.
Definition subst_equiv_property := forall x1 x2 a1 a2,
  cequiv <{ x1 := a1; x2 := a2 }>
         <{ x1 := a1; x2 := subst_aexp x1 a1 a2 }>.
```

该命题成立吗?

内联变量

- 不成立：因为程序有副作用，同样表达式两次求职并不一定相等
  - X := X + 1; Y := X
  - X := X + 1; Y := X + 1
- 内联变量
  - ```
    Theorem subst_inequiv :
      ~ subst_equiv_property.
    ```
  - 用上述反例可以证明
    ```

作业

- 完成Equiv中standard非optional并不属于Additional Exercises的习题
  - 请使用最新英文版教材