CS720

Logical Foundations of Computer Science

Lecture 13: Program equivalence

Tiago Cogumbreiro

`Imp.v`

Due Thursday October 18, 11:59pm EST

IndProp.v

Due Friday October 19, 11:59pm EST

`Equiv.v`

Due Thursday October 25, 11:59pm EST

Programming Language Foundation

Volume 2 of Software Foundations

Summary

- Behavioral equivalence
- Properties on behavioral equivalence
- Program transformations

Program equivalence

- A framework to compare "equivalent" programs, notation $P \equiv Q$
- The notion of equivalent is generic
- Program equivalence can be used to reason about correctness of algorithms
- Program equivalence can be used to reason about the correctness of program transformations

Examples:

- compilable programs
- programs that produce the same output
- programs that perform the same assignments
- programs that read the same variables

Usual equivalence properties

- Reflexive: $P \equiv P$
- Symmetric: $P \equiv Q \implies Q \equiv P$
- Transitive: $P \equiv Q \implies Q \equiv R \implies P \equiv R$
- Congruence: $P \equiv Q \implies \mathcal{C}(P) \equiv \mathcal{C}(Q)$ where $\mathcal{C} : \mathcal{P} \to \mathcal{P}$ is known as a *context*, a program with a "whole" that is filled with the input program, outputting a "complete" program; it is expected that the input occurs in the output.

Syntactic equivalence

> If two programs are textually equal (are the same syntactic term), then we say that the two programs are syntactically equivalent.

Example: `APlus (ANum 3) (ANum 0)` is syntactically equivalent to `APlus (ANum 3) (ANum 0)`.

Behavioral equivalence

> If two programs start from an initial state and reach the same final state, then we say that the two programs are behaviorally equivalent.

Example:

```
X:=3;; WHILE 1≤X DO Y:=Y+1;; X:=X-1 END
```

is behaviorally equivalent to

```
X:=0 ;; Y:=3
```

How do we formalize behavioral equivalence for arithmetic expressions, boolean expressions, commands?

Behavioral equivalence

For arithmetic expressions $a_1 \equiv a_2$, e.g., $x - x \equiv \mathtt{0}$:

Behavioral equivalence

For arithmetic expressions $a_1 \equiv a_2$, e.g., $x - x \equiv \mathtt{0}$:

$$\frac{\forall s\colon \mathtt{aeval}(s, a_1) = \mathtt{aeval}(s, a_2)}{a_1 \equiv a_2}$$

For boolean expressions $b_1 \equiv b_2$, e.g., $(x - x = 0) \equiv \top$:

Behavioral equivalence

For arithmetic expressions $a_1 \equiv a_2$, e.g., $x - x \equiv \mathtt{0}$:

$$\frac{\forall s \colon \mathtt{aeval}(s, a_1) = \mathtt{aeval}(s, a_2)}{a_1 \equiv a_2}$$

For boolean expressions $b_1 \equiv b_2$, e.g., $(x - x = 0) \equiv \top$:

$$\frac{\forall s \colon \mathtt{beval}(s, b_1) = \mathtt{beval}(s, b_2)}{b_1 \equiv b_2}$$

For commands $c_1 \equiv c_2$:

Behavioral equivalence

For arithmetic expressions $a_1 \equiv a_2$, e.g., $x - x \equiv \mathtt{0}$:

$$\frac{\forall s\colon \mathtt{aeval}(s, a_1) = \mathtt{aeval}(s, a_2)}{a_1 \equiv a_2}$$

For boolean expressions $b_1 \equiv b_2$, e.g., $(x - x = 0) \equiv \top$:

$$\frac{\forall s\colon \mathtt{beval}(s, b_1) = \mathtt{beval}(s, b_2)}{b_1 \equiv b_2}$$

For commands $c_1 \equiv c_2$:

$$\frac{\forall s_1, \forall s_2\colon c_1 \;/\; s_1 \;\backslash\backslash\; s_2 \iff c_2 \;/\; s_1 \;\backslash\backslash\; s_2}{c_1 \equiv c_2}$$

Exercise: skip

Prove that

$$\texttt{SKIP};; c \equiv c$$

```
Theorem skip_left: forall c,
  cequiv (SKIP;; c) c.
```

Exercise: if

If $b \equiv \top$, then $\texttt{IFB}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2\ \texttt{FI} \equiv c_1$.

```
Theorem IFB_true: forall b c1 c2,
     bequiv b BTrue  ->
     cequiv (IFB b THEN c1 ELSE c2 FI) c1.
```

> What could b in $b \equiv \top$ be? For instance, the following statement holds. (By using lemmas `Nat.add_0_r`, `Nat.eqb_refl`.)

$$(x + x = 2 * x) \equiv \top$$

```
Require Import PeanoNat.
Goal forall x, bequiv (x + x = 2 * x) BTrue.
```

Exercise: while

A similar result to `IFB_true` is the following.

Theorem: If $b \equiv \bot$, then $\texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END} \equiv \texttt{SKIP}$.

A more interesting result to show is.

Theorem: If $b \equiv \top$, then $\neg\texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END}\ /\ s \setminus\!\setminus s'$.

```
Lemma WHILE_true_nonterm : forall b c st st',
  bequiv b BTrue ->
  ~( (WHILE b DO c END) / st \\ st' ).
Proof.
  intros b c st st' Hb.
  intros H.
  remember (WHILE b DO c END) as cw eqn:Heqcw.
  induction H.
```

A note on proving reduction by induction

```
1 subgoal
b : bexp
c : com
st, st' : state
Hb : bequiv b BTrue
H : (WHILE b DO c END) / st \\ st'
______________________________________(1/1)
False
```

> Notice how induction on `c` does very little, we need to reason about the *derivation tree* of reduction and get the induction principle from `H`. Whenever we need to reason about all possible derivation trees (say H), it is crucial to:
>
> 1. `remember` the expression that is getting "smaller" (say WHILE b DO c END) before performing induction, and then
> 2. invert the equation that results from `remember`.

Loop unrolling

A common code transformation performed by compilers can be proved correct:

Theorem:

$$\texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END} \equiv \texttt{IFB}\ b\ \texttt{THEN}\ (c;;\texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END})\ \texttt{ELSE}\ \texttt{SKIP}\ \texttt{FI}$$

```
Theorem loop_unrolling: forall b c,
  cequiv
    (WHILE b DO c END)
    (IFB b THEN (c ;; WHILE b DO c END) ELSE SKIP FI).
```

(Proof in the book.)

Properties of equivalences

An equivalence relation is:

- reflexive
- symmetric
- transitive

Show that `aquiv`, `bequiv`, and `cequiv` each is an equivalence relation.

```
Lemma refl_cequiv : forall (c : com), cequiv c c.
Lemma sym_cequiv : forall (c1 c2 : com), cequiv c1 c2 → cequiv c2 c1.
Lemma trans_cequiv : forall (c1 c2 c3 : com), cequiv c1 c2 → cequiv c2 c3 → cequiv c1 c3.
```

$\equiv$ is a congruence

Generally a congruence can be described as

$$c \equiv c' \implies \mathcal{C}(c) \equiv \mathcal{C}(c')$$

For commands this corresponds to proving

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

$$\frac{b \equiv b' \qquad c_1 \equiv c_1' \qquad c_2 \equiv c_2'}{\texttt{IFB}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2\ \texttt{FI} \equiv \texttt{IFB}\ b'\ \texttt{THEN}\ c_1'\ \texttt{ELSE}\ c_2'\ \texttt{FI}}$$

$$\frac{b \equiv b' \qquad c \equiv c'}{\texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END} \equiv \texttt{WHILE}\ b'\ \texttt{DO}\ c'\ \texttt{END}}$$

Example: congruence on while

```
Theorem CWhile_congruence : forall b1 b1' c1 c1',
  bequiv b1 b1' → cequiv c1 c1' →
  cequiv (WHILE b1 DO c1 END) (WHILE b1' DO c1' END).
Proof.
  unfold bequiv,cequiv.
  intros b1 b1' c1 c1' Hb1e Hc1e st st'.
  split; intros Hce.
```

Example: congruence on while

```
Theorem CWhile_congruence : forall b1 b1' c1 c1',
  bequiv b1 b1' → cequiv c1 c1' →
  cequiv (WHILE b1 DO c1 END) (WHILE b1' DO c1' END).
Proof.
  unfold bequiv,cequiv.
  intros b1 b1' c1 c1' Hb1e Hc1e st st'.
  split; intros Hce.
```

> **See slide 15.** To solve each side, we need to somehow simplify our hypothesis. If we try to do the proof by induction on the structure of `c1`, we quickly see that the induction hypothesis is unhelpful to our goal. We need to reason about all possible derivation trees, so that we can capture all possible loop executions. It is crucial to remember `WHILE b1 DO c1 END` before performing induction, otherwise we loose that information and know nothing about the structure of `WHILE b1 DO c1 END` and how it relates to the derivation tree.

Revisiting code transformations

```
Fixpoint fold_constants_com (c : com) : com :=
  match c with
  | SKIP ⇒ SKIP
  | i ::= a  ⇒ CAss i (fold_constants_aexp a)
  | c1 ;; c2  ⇒ (fold_constants_com c1) ;; (fold_constants_com c2)
  | IFB b THEN c1 ELSE c2 FI ⇒
      match fold_constants_bexp b with
      | BTrue ⇒ fold_constants_com c1
      | BFalse ⇒ fold_constants_com c2
      | b' ⇒ IFB b' THEN fold_constants_com c1
                     ELSE fold_constants_com c2 FI
      end
  | WHILE b DO c END ⇒
      match fold_constants_bexp b with
      | BTrue ⇒ WHILE BTrue DO SKIP END
      | BFalse ⇒ SKIP
      | b' ⇒ WHILE b' DO (fold_constants_com c) END
      end
  end.
```

Code transformations (and congruence)

Theorem: `fold_constants_com` is sound (that is, the optimized code is behaviorally equivalent to the original code).

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

Theorem fold_constants_com_sound :  ctrans_sound fold_constants_com.
```

Summary

- Behavioral equivalence
- Properties on behavioral equivalence
- Program transformations
- Induction on derivation trees