Type Theory and Coq

Small step and Auto

Zhuoran Liu

May 30th

Radboud University

Difference between small step and big step

The big step specify how a given expression can be evaluated to its final value. The style is simple and natural for many purposes and is called natural semantics. But it does not include the intermediate states that it passes through along the way. And it is more technical, but critical in some situations.

So, for lots of reasons, we'd like to have a finer-grained way of defining and reasoning about program behaviors. We replace the big-step eval relation with a small-step relation that specifies, for a given program, how the "atomic steps" of computation are performed.

A Toy Language

```
Inductive tm : Type :=
  | C : nat → tm (* Constant *)
  | P : tm → tm → tm. (* Plus *)


Tactic Notation "tm_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "C" | Case_aux c "P" ].
```

$$\frac{}{\texttt{C n} \Downarrow \texttt{n}}\ (\text{E\_Const})$$

$$\frac{\texttt{t}_1 \Downarrow \texttt{n1} \qquad \texttt{t}_2 \Downarrow \texttt{n2}}{\texttt{P t}_1\ \texttt{t}_2 \Downarrow \texttt{C (n1 + n2)}}\ (\text{E\_Plus})$$

$$\frac{}{\texttt{P (C n1) (C n2)} \Rightarrow \texttt{C (n1 + n2)}}\ (\text{ST\_PlusConstConst})$$

$$\frac{\texttt{t}_1 \Rightarrow \texttt{t}_1'}{\texttt{P t}_1\ \texttt{t}_2 \Rightarrow \texttt{P t}_1'\ \texttt{t}_2}\ (\text{ST\_Plus1})$$

$$\frac{\texttt{t}_2 \Rightarrow \texttt{t}_2'}{\texttt{P (C n1) t}_2 \Rightarrow \texttt{P (C n1) t}_2'}\ (\text{ST\_Plus2})$$

Evaluator

```
Inductive step : tm → tm → Prop :=
  | ST_PlusConstConst : ∀n1 n2,
       P (C n1) (C n2) ⇒ C (n1 + n2)
  | ST_Plus1 : ∀t1 t1' t2,
       t1 ⇒ t1' →
       P t1 t2 ⇒ P t1' t2
  | ST_Plus2 : ∀n1 t2 t2',
       t2 ⇒ t2' →
       P (C n1) t2 ⇒ P (C n1) t2'

  where " t '⇒' t' " := (step t t').
```

Notice:

1. Defining single reduction step.
2. Consider the leftmost P node and rewrite.
3. Constant term cannot take a step.

Example. test_step_2.

Relations

A relation on a set X is a family of propositions parameterized by two elements of X.

Definition relation (X: Type) := X→X→Prop.

Theorem: For each t, there is at most one t' such that t steps to t' (t ⇒ t' is provable). Formally, this is the same as saying that ⇒ is deterministic.

Proof sketch.
1. if both are ST_PlusConstConst, the result is immediate.
2. The cases when both derivations end with ST_Plus1 or ST_Plus2 follow by the induction hypothesis.
3. It cannot happen that one is ST_PlusConstConst and the other is ST_Plus1 or ST_Plus2, since this would imply that x has the form P t1 t2 where both t1 and t2 are constants (by ST_PlusConstConst) and one of t1 or t2 has the form P ....
4.Similarly, it cannot happen that one is ST_Plus1 and the other is ST_Plus2, since this would imply that x has the form P t1 t2 where t1 has both the form P t1 t2 and the form C n

Example. step_deterministic.

Values

Intuitively, it is clear that the final states of the machine are always terms of the form C n for some n. We call such terms values.

```
Inductive value : tm → Prop :=
  v_const : ∀n, value (C n).
```

```
Inductive step : tm → tm → Prop :=
  | ST_PlusConstConst : ∀n1 n2,
          P (C n1) (C n2)
       ⇒ C (n1 + n2)
  | ST_Plus1 : ∀t1 t1' t2,
       t1 ⇒ t1' →
       P t1 t2 ⇒ P t1' t2
  | ST_Plus2 : ∀v1 t2 t2',
       value v1 →
       t2 ⇒ t2' →
       P v1 t2 ⇒ P v1 t2'
```

$$\frac{}{\texttt{P (C n1) (C n2)} \Rightarrow \texttt{C (n1 + n2)}}\ \text{(ST\_PlusConstConst)}$$

$$\frac{\texttt{t}_1 \Rightarrow \texttt{t}_1'}{\texttt{P t}_1\ \texttt{t}_2 \Rightarrow \texttt{P t}_1'\ \texttt{t}_2}\ \text{(ST\_Plus1)}$$

$$\frac{\texttt{value v}_1 \qquad \texttt{t}_2 \Rightarrow \texttt{t}_2'}{\texttt{P v}_1\ \texttt{t}_2 \Rightarrow \texttt{P v}_1\ \texttt{t}_2'}\ \text{(ST\_Plus2)}$$

Strong Progress

Theorem (Strong Progress): If t is a term, then either t is a value, or there exists a term t' such that t ⇒ t'.

Proof: By induction on t.
Suppose t = C n. Then t is a value.
Suppose t = P t1 t2, where (by the IH) t1 is either a value or can step to some t1', and where t2 is either a value or can step to some t2'. We must show P t1 t2 is either a value or steps to some t'.
If t1 and t2 are both values, then t can take a step, by ST_PlusConstConst.
If t1 is a value and t2 can take a step, then so can t, by ST_Plus2.
If t1 can take a step, then so can t, by ST_Plus1.

Example. strong_progress.

Normal Form

We call terms that cannot make progress normal form.

```
Definition normal_form {X:Type} (R:relation X) (t:X) : Prop :=
  ¬ ∃t', R t t'.

Lemma value_is_nf : ∀v,
  value v → normal_form step v.

Lemma nf_is_value : ∀t,
  normal_form step t → value t.
```

```
Corollary nf_same_as_value : ∀t,
  normal_form step t ↔ value t.
```

Value is a syntactic concept — it is defined by looking at the form of a term — while normal_form is a semantic one — it is defined by looking at how the term steps.

Example. nf_same_as_value.

Multi-Step Reduction

Given a relation R, we define a relation multi R, called the multi-step closure of R as follows:

```
Inductive multi {X:Type} (R: relation X) : relation X :=
  | multi_refl : ∀(x : X), multi R x x
  | multi_step : ∀(x y z : X),
                    R x y →
                    multi R y z →
                    multi R x z.
```

```
Tactic Notation "multi_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "multi_refl" | Case_aux c "multi_step" ].
```

Properties

1. Reflexive
2. Single-step executions are a particular case of multi-step executions.
3. Transitive

Example. test_multistep_1.

Normal Form

If t reduces to t' in zero or more steps and t' is a normal form, we say that "t' is a normal form of t."

Definition step_normal_form := normal_form step.

Definition normal_form_of (t t' : tm) :=
  (t ⇒* t' ∧ step_normal_form t').

Theorem: The step function is normalizing — i.e., for every t there exists some t' such that t steps to t' and t' is a normal form.

Example. step_normalizing : normalizing step.

Equivalence

Theorem eval__multistep : ∀t n,
  t ⇓ n → t ⇒* C n.

```
P t1 t2 ⇒                (by ST_Plus1)
P t1' t2 ⇒               (by ST_Plus1)
P t1'' t2 ⇒              (by ST_Plus1)
...
P (C n1) t2 ⇒            (by ST_Plus2)
P (C n1) t2' ⇒           (by ST_Plus2)
P (C n1) t2'' ⇒          (by ST_Plus2)
...
P (C n1) (C n2) ⇒        (by ST_PlusConstConst)
C (n1 + n2)
```

Rest of Small step

Small-Step Imp

Concurrent Imp

A Small-Step Stack Machine

Auto

We always wrote proof scripts that apply relevant hypotheses or lemmas by name. In particular, when a chain of hypothesis applications is needed, we have specified them explicitly.

The auto tactic frees us from this drudgery by searching for a sequence of applications that will prove the goal.

The auto tactic solves goals that are solvable by any combination of intros and apply.

Properties

1. Safe.
2. The search limit can be set.
3. Consider the hypotheses in the current context together with a hint database.
4. Use info_auto to check the facts using.
5. extend the hint database by auto using.
6. We can add constructors and lemmas to the global hint database.

Example. Auto.v

Example. ceval_deterministic

```
Theorem ceval_deterministic: ∀c st st1 st2,
     c / st ⇓ st1 →
     c / st ⇓ st2 →
     st1 = st2.
```

Searching Hypotheses

Tackling the contradiction cases. Each of them occurs in a situation where we have both

H1: beval st b = false
and
H2: beval st b = true

as hypotheses. The contradiction is evident, but demonstrating it is a little complicated: we have to locate the two hypotheses H1 and H2 and do a rewrite following by an inversion.

Process

Ltac inv H := inversion H; subst; clear H.

We can abstract out the piece of script in question by writing a small amount of paramerized Ltac.

Ltac rwinv H1 H2 := rewrite H1 in H2; inv H2.

We really want Coq to discover the relevant hypotheses for us. We can do this by using the match goal with ... end facility of Ltac.

```
Ltac find_rwinv :=
  match goal with
    H1: ?E = true, H2: ?E = false ⊢ _ ⇒ rwinv H1 H2
  end.

Example. ceval_deterministic'''.
```

Process

Finally, let's see about the remaining cases. Each of them involves applying a conditional hypothesis to extract an equality. Currently we have phrased these as assertions, so that we have to predict what the resulting equality will be (although we can then use auto to prove it.) An alternative is to pick the relevant hypotheses to use, and then rewrite with them, as follows:

```
Theorem ceval_deterministic'''': ∀c st st1 st2,
     c / st ⇓ st1 →
     c / st ⇓ st2 →
     st1 = st2.
Proof.
  intros c st st1 st2 E1 E2;
  generalize dependent st2;
  ceval_cases (induction E1) Case;
          intros st2 E2; inv E2; try find_rwinv; auto.
  Case "E_Seq".
    rewrite (IHE1_1 st'0 H1) in ×. auto.
  Case "E_WhileLoop".
    SCase "b evaluates to true".
      rewrite (IHE1_1 st'0 H3) in ×. auto. Qed.
```

```
Ltac find_eqn :=
  match goal with
    H1: forall x, ?P x -> ?L = ?R, H2: ?P ?X |- _ =>
          rewrite (H1 X H2) in *
```

Repeat

The big pay-off in this approach is that our proof script should be robust in the face of modest changes to our language. For example, we can add a REPEAT command to the language.

REPEAT behaves like WHILE, except that the loop guard is checked after each execution of the body, with the loop repeating as long as the guard stays false. Because of this, the body will always execute at least once.

Example. ceval_deterministic'.