SOFTWARE FOUNDATIONS

VOLUME 2: PROGRAMMING LANGUAGE FOUNDATIONS

TABLE OF CONTENTS INDEX ROADMAP

TYPES

TYPE SYSTEMS

Our next major topic is *type systems* — static program analyses that classify expressions according to the "shapes" of their results. We'll begin with a typed version of the simplest imaginable language, to introduce the basic ideas of types and typing rules and the fundamental theorems about type systems: *type preservation* and *progress*. In chapter Stlc we'll move on to the *simply typed lambda-calculus*, which lives at the core of every modern functional programming language (including Coq!).

```
Set Warnings "-notation-overridden,-parsing".
Require Import Coq.Arith.Arith.
Require Import Maps.
Require Import Imp.
Require Import Smallstep.

Hint Constructors multi.
```

Typed Arithmetic Expressions

To motivate the discussion of type systems, let's begin as usual with a tiny toy language. We want it to have the potential for programs to go wrong because of runtime type errors, so we need something a tiny bit more complex than the language of constants and addition that we used in chapter Smallstep: a single kind of data (e.g., numbers) is too simple, but just two kinds (numbers and booleans) gives us enough material to tell an interesting story.

The language definition is completely routine.

Syntax

Here is the syntax, informally:

```
t ::= true
    | false
```

```
  | if t then t else t
  | 0
  | succ t
  | pred t
  | iszero t
```

And here it is formally:

```
Inductive tm : Type :=
  | ttrue : tm
  | tfalse : tm
  | tif : tm → tm → tm → tm
  | tzero : tm
  | tsucc : tm → tm
  | tpred : tm → tm
  | tiszero : tm → tm.
```

Values are `true`, `false`, and numeric values...

```
Inductive bvalue : tm → Prop :=
  | bv_true : bvalue ttrue
  | bv_false : bvalue tfalse.

Inductive nvalue : tm → Prop :=
  | nv_zero : nvalue tzero
  | nv_succ : ∀ t, nvalue t → nvalue (tsucc t).

Definition value (t:tm) := bvalue t ∨ nvalue t.

Hint Constructors bvalue nvalue.
Hint Unfold value.
Hint Unfold update.
```

Operational Semantics

Here is the single-step relation, first informally...

$$\frac{}{\texttt{if true then } t_1 \texttt{ else } t_2 \texttt{ ==> } t_1} \quad \text{(ST\_IfTrue)}$$

$$\frac{}{\texttt{if false then } t_1 \texttt{ else } t_2 \texttt{ ==> } t_2} \quad \text{(ST\_IfFalse)}$$

$$\frac{t_1 \texttt{ ==> } t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \texttt{ ==> if } t_1' \texttt{ then } t_2 \texttt{ else } t_3} \quad \text{(ST\_If)}$$

$$\frac{t_1 \texttt{ ==> } t_1'}{\texttt{succ } t_1 \texttt{ ==> succ } t_1'} \quad \text{(ST\_Succ)}$$

$$\frac{}{\texttt{pred 0 ==> 0}} \quad \text{(ST\_PredZero)}$$

$$\frac{\texttt{numeric value } v_1}{} \quad \text{(ST\_PredSucc)}$$

$$\texttt{pred (succ } v_1\texttt{) ==> } v_1$$

$$\frac{t_1 \texttt{ ==> } t_1'}{\texttt{pred } t_1 \texttt{ ==> pred } t_1'} \text{ (ST\_Pred)}$$

$$\frac{}{\texttt{iszero 0 ==> true}} \text{ (ST\_IszeroZero)}$$

$$\frac{\texttt{numeric value } v_1}{\texttt{iszero (succ } v_1\texttt{) ==> false}} \text{ (ST\_IszeroSucc)}$$

$$\frac{t_1 \texttt{ ==> } t_1'}{\texttt{iszero } t_1 \texttt{ ==> iszero } t_1'} \text{ (ST\_Iszero)}$$

... and then formally:

```
Reserved Notation "t1 '==>' t2" (at level 40).

Inductive step : tm → tm → Prop :=
  | ST_IfTrue : ∀ t1 t2,
      (tif ttrue t1 t2) ==> t1
  | ST_IfFalse : ∀ t1 t2,
      (tif tfalse t1 t2) ==> t2
  | ST_If : ∀ t1 t1' t2 t3,
      t1 ==> t1' →
      (tif t1 t2 t3) ==> (tif t1' t2 t3)
  | ST_Succ : ∀ t1 t1',
      t1 ==> t1' →
      (tsucc t1) ==> (tsucc t1')
  | ST_PredZero :
      (tpred tzero) ==> tzero
  | ST_PredSucc : ∀ t1,
      nvalue t1 →
      (tpred (tsucc t1)) ==> t1
  | ST_Pred : ∀ t1 t1',
      t1 ==> t1' →
      (tpred t1) ==> (tpred t1')
  | ST_IszeroZero :
      (tiszero tzero) ==> ttrue
  | ST_IszeroSucc : ∀ t1,
       nvalue t1 →
      (tiszero (tsucc t1)) ==> tfalse
  | ST_Iszero : ∀ t1 t1',
      t1 ==> t1' →
      (tiszero t1) ==> (tiszero t1')

where "t1 '==>' t2" := (step t1 t2).
```

```
Hint Constructors step.
```

Notice that the `step` relation doesn't care about whether expressions make global sense — it just checks that the operation in the *next* reduction step is being applied to the right kinds of operands. For example, the term `succ true` (i.e., `tsucc ttrue` in the formal syntax) cannot take a step, but the almost as obviously nonsensical term

```
succ (if true then true else true)
```

can take a step (once, before becoming stuck).

Normal Forms and Values

The first interesting thing to notice about this `step` relation is that the strong progress theorem from the Smallstep chapter fails here. That is, there are terms that are normal forms (they can't take a step) but not values (because we have not included them in our definition of possible "results of reduction"). Such terms are *stuck*.

```
Notation step_normal_form := (normal_form step).

Definition stuck (t:tm) : Prop :=
  step_normal_form t ∧ ¬ value t.

Hint Unfold stuck.
```

Exercise: 2 stars (some_term_is_stuck)

```
Example some_term_is_stuck :
  ∃ t, stuck t.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

However, although values and normal forms are *not* the same in this language, the set of values is included in the set of normal forms. This is important because it shows we did not accidentally define things so that some value could still take a step.

Exercise: 3 stars (value_is_nf)

```
Lemma value_is_nf : ∀ t,
  value t → step_normal_form t.
+
```

(Hint: You will reach a point in this proof where you need to use an induction to reason about a term that is known to be a numeric value. This induction can be performed either over the term itself or over the evidence that it is a numeric value. The proof goes through in either case, but you will find that one way is quite a bit shorter than the other. For the sake of the exercise, try to complete the proof both ways.) □

Exercise: 3 stars, optional (step_deterministic)

Use `value_is_nf` to show that the `step` relation is also deterministic.

```
Theorem step_deterministic:
  deterministic step.
Proof with eauto.
  (* FILL IN HERE *) Admitted.
```

□

Typing

The next critical observation is that, although this language has stuck terms, they are always nonsensical, mixing booleans and numbers in a way that we don't even *want* to have a meaning. We can easily exclude such ill-typed terms by defining a *typing relation* that relates terms to the types (either numeric or boolean) of their final results.

```
Inductive ty : Type :=
  | TBool : ty
  | TNat : ty.
```

In informal notation, the typing relation is often written `|- t ∈ T` and pronounced "`t` has type `T`." The `|-` symbol is called a "turnstile." Below, we're going to see richer typing relations where one or more additional "context" arguments are written to the left of the turnstile. For the moment, the context is always empty.

$$\frac{}{\vdash \texttt{true} \in \texttt{Bool}} \text{ (T\_True)}$$

$$\frac{}{\vdash \texttt{false} \in \texttt{Bool}} \text{ (T\_False)}$$

$$\frac{\vdash t_1 \in \texttt{Bool} \quad \vdash t_2 \in T \quad \vdash t_3 \in T}{\vdash \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \in T} \text{ (T\_If)}$$

$$\frac{}{\vdash 0 \in \texttt{Nat}} \text{ (T\_Zero)}$$

$$\frac{\vdash t_1 \in \texttt{Nat}}{\vdash \texttt{succ } t_1 \in \texttt{Nat}} \text{ (T\_Succ)}$$

$$\frac{\vdash t_1 \in \texttt{Nat}}{\vdash \texttt{pred } t_1 \in \texttt{Nat}} \text{ (T\_Pred)}$$

$$\frac{\vdash t_1 \in \texttt{Nat}}{\vdash \texttt{iszero } t_1 \in \texttt{Bool}} \text{ (T\_IsZero)}$$

```
Reserved Notation "'|-' t '∈' T" (at level 40).

Inductive has_type : tm → ty → Prop :=
  | T_True :
       |- ttrue ∈ TBool
  | T_False :
       |- tfalse ∈ TBool
  | T_If : ∀ t1 t2 t3 T,
```

```
          |- t1 ∈ TBool →
          |- t2 ∈ T →
          |- t3 ∈ T →
          |- tif t1 t2 t3 ∈ T
    | T_Zero :
          |- tzero ∈ TNat
    | T_Succ : ∀ t1,
          |- t1 ∈ TNat →
          |- tsucc t1 ∈ TNat
    | T_Pred : ∀ t1,
          |- t1 ∈ TNat →
          |- tpred t1 ∈ TNat
    | T_Iszero : ∀ t1,
          |- t1 ∈ TNat →
          |- tiszero t1 ∈ TBool

  where "'|-' t '∈' T" := (has_type t T).

  Hint Constructors has_type.

  Example has_type_1 :
    |- tif tfalse tzero (tsucc tzero) ∈ TNat.
  Proof.
    apply T_If.
      - apply T_False.
      - apply T_Zero.
      - apply T_Succ.
        + apply T_Zero.
  Qed.
```

(Since we've included all the constructors of the typing relation in the hint database, the `auto` tactic can actually find this proof automatically.)

It's important to realize that the typing relation is a *conservative* (or *static*) approximation: it does not consider what happens when the term is reduced — in particular, it does not calculate the type of its normal form.

```
  Example has_type_not :
    ¬ (|- tif tfalse tzero ttrue ∈ TBool).
  +
```

Exercise: 1 star, optional (succ_hastype_nat__hastype_nat)

```
  Example succ_hastype_nat__hastype_nat : ∀ t,
    |- tsucc t ∈ TNat →
    |- t ∈ TNat.
  Proof.
    (* FILL IN HERE *) Admitted.
```

□

Canonical forms

The following two lemmas capture the fundamental property that the definitions of boolean and numeric values agree with the typing relation.

```
Lemma bool_canonical : ∀ t,
  |- t ∈ TBool → value t → bvalue t.
+

Lemma nat_canonical : ∀ t,
  |- t ∈ TNat → value t → nvalue t.
+
```

Progress

The typing relation enjoys two critical properties. The first is that well-typed normal forms are not stuck — or conversely, if a term is well typed, then either it is a value or it can take at least one step. We call this *progress*.

Exercise: 3 stars (finish_progress)

```
Theorem progress : ∀ t T,
  |- t ∈ T →
  value t ∨ ∃ t', t ==> t'.
```

Complete the formal proof of the `progress` property. (Make sure you understand the parts we've given of the informal proof in the following exercise before starting — this will save you a lot of time.)

```
+
```

□

Exercise: 3 stars, advanced (finish_progress_informal)

Complete the corresponding informal proof:

Theorem: If $|- t \in T$, then either t is a value or else $t ==> t'$ for some t'.

Proof: By induction on a derivation of $|- t \in T$.

- If the last rule in the derivation is `T_If`, then $t = \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3$, with $|- t_1 \in \texttt{Bool}$, $|- t_2 \in T$ and $|- t_3 \in T$. By the IH, either t_1 is a value or else t_1 can step to some t_1'.
  - If t_1 is a value, then by the canonical forms lemmas and the fact that $|- t_1 \in \texttt{Bool}$ we have that t_1 is a `bvalue` — i.e., it is either `true` or `false`. If $t_1 = \texttt{true}$, then t steps to t_2 by `ST_IfTrue`, while if $t_1 = \texttt{false}$, then t steps to t_3 by `ST_IfFalse`. Either way, t can step, which is what we wanted to show.
  - If t_1 itself can take a step, then, by `ST_If`, so can t.
- `(* FILL IN HERE *)`

□

This theorem is more interesting than the strong progress theorem that we saw in the Smallstep chapter, where *all* normal forms were values. Here a term can be stuck, but only if it is ill typed.

Type Preservation

The second critical property of typing is that, when a well-typed term takes a step, the result is also a well-typed term.

Exercise: 2 stars (finish_preservation)

```
Theorem preservation : ∀ t t' T,
  |- t ∈ T →
  t ==> t' →
  |- t' ∈ T.
```

Complete the formal proof of the `preservation` property. (Again, make sure you understand the informal proof fragment in the following exercise first.)

```
  +
□
```

Exercise: 3 stars, advanced (finish_preservation_informal)

Complete the following informal proof:

Theorem: If $|- t \in T$ and $t ==> t'$, then $|- t' \in T$.

Proof: By induction on a derivation of $|- t \in T$.

- If the last rule in the derivation is `T_If`, then $t = \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3$, with $|- t_1 \in \texttt{Bool}$, $|- t_2 \in T$ and $|- t_3 \in T$.

  Inspecting the rules for the small-step reduction relation and remembering that t has the form `if ...`, we see that the only ones that could have been used to prove $t ==> t'$ are `ST_IfTrue`, `ST_IfFalse`, or `ST_If`.

  - If the last rule was `ST_IfTrue`, then $t' = t_2$. But we know that $|- t_2 \in T$, so we are done.
  - If the last rule was `ST_IfFalse`, then $t' = t_3$. But we know that $|- t_3 \in T$, so we are done.
  - If the last rule was `ST_If`, then $t' = \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3$, where $t_1 ==> t_1'$. We know $|- t_1 \in \texttt{Bool}$ so, by the IH, $|- t_1' \in \texttt{Bool}$. The `T_If` rule then gives us $|- \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3 \in T$, as required.

- `(* FILL IN HERE *)`

□

Exercise: 3 stars (preservation_alternate_proof)

Now prove the same property again by induction on the *evaluation* derivation instead of on the typing derivation. Begin by carefully reading and thinking about the first few lines of the above proofs to make sure you understand what each one is doing. The set-up for this proof is similar, but not exactly the same.

```
Theorem preservation' : ∀ t t' T,
  |- t ∈ T →
  t ==> t' →
  |- t' ∈ T.
Proof with eauto.
  (* FILL IN HERE *) Admitted.
```

□

The preservation theorem is often called *subject reduction*, because it tells us what happens when the "subject" of the typing relation is reduced. This terminology comes from thinking of typing statements as sentences, where the term is the subject and the type is the predicate.

Type Soundness

Putting progress and preservation together, we see that a well-typed term can never reach a stuck state.

```
Definition multistep := (multi step).
Notation "t1 '==>*' t2" := (multistep t1 t2) (at level 40).

Corollary soundness : ∀ t t' T,
  |- t ∈ T →
  t ==>* t' →
  ~(stuck t').
```

+

Aside: the `normalize` Tactic

When experimenting with definitions of programming languages in Coq, we often want to see what a particular concrete term steps to — i.e., we want to find proofs for goals of the form `t ==>* t'`, where `t` is a completely concrete term and `t'` is unknown. These proofs are quite tedious to do by hand. Consider, for example, reducing an arithmetic expression using the small-step relation `astep`.

```
Module NormalizePlayground.
Import Smallstep.

Example step_example1 :
  (P (C 3) (P (C 3) (C 4)))
  ==>* (C 10).
Proof.
  apply multi_step with (P (C 3) (C 7)).
    apply ST_Plus2.
      apply v_const.
      apply ST_PlusConstConst.
  apply multi_step with (C 10).
```

```
    apply ST_PlusConstConst.
  apply multi_refl.
Qed.
```

The proof repeatedly applies `multi_step` until the term reaches a normal form. Fortunately The sub-proofs for the intermediate steps are simple enough that `auto`, with appropriate hints, can solve them.

```
Hint Constructors step value.
Example step_example1' :
  (P (C 3) (P (C 3) (C 4)))
  ==>* (C 10).
Proof.
  eapply multi_step. auto. simpl.
  eapply multi_step. auto. simpl.
  apply multi_refl.
Qed.
```

The following custom `Tactic Notation` definition captures this pattern. In addition, before each step, we print out the current goal, so that we can follow how the term is being reduced.

```
Tactic Notation "print_goal" :=
  match goal with |- ?x ⇒ idtac x end.

Tactic Notation "normalize" :=
  repeat (print_goal; eapply multi_step ;
            [ (eauto 10; fail) | (instantiate; simpl)]);
  apply multi_refl.

Example step_example1'' :
  (P (C 3) (P (C 3) (C 4)))
  ==>* (C 10).
Proof.
  normalize.
  (* The print_goal in the normalize tactic shows
     a trace of how the expression reduced...
         (P (C 3) (P (C 3) (C 4)) ==>* C 10)
         (P (C 3) (C 7) ==>* C 10)
         (C 10 ==>* C 10)
  *)
Qed.
```

The `normalize` tactic also provides a simple way to calculate the normal form of a term, by starting with a goal with an existentially bound variable.

```
Example step_example1''' : ∃ e',
  (P (C 3) (P (C 3) (C 4)))
  ==>* e'.
Proof.
  eapply ex_intro. normalize.
(* This time, the trace is:
       (P (C 3) (P (C 3) (C 4)) ==>* ?e')
       (P (C 3) (C 7) ==>* ?e')
       (C 10 ==>* ?e')
```

```
     where ?e' is the variable ``guessed'' by eapply. *)
  Qed.
```

Exercise: 1 star (normalize_ex)

```
  Theorem normalize_ex : ∃ e',
    (P (C 3) (P (C 2) (C 1)))
    ==>* e'.
  Proof.
    (* FILL IN HERE *) Admitted.
```

□

Exercise: 1 star, optional (normalize_ex')

For comparison, prove it using `apply` instead of `eapply`.

```
  Theorem normalize_ex' : ∃ e',
    (P (C 3) (P (C 2) (C 1)))
    ==>* e'.
  Proof.
    (* FILL IN HERE *) Admitted.
```

□

```
  End NormalizePlayground.

  Tactic Notation "print_goal" :=
    match goal with |- ?x ⇒ idtac x end.
  Tactic Notation "normalize" :=
    repeat (print_goal; eapply multi_step ;
              [ (eauto 10; fail) | (instantiate; simpl)]);
    apply multi_refl.
```

Additional Exercises

Exercise: 2 stars, recommended (subject_expansion)

Having seen the subject reduction property, one might wonder whether the opposity property — subject *expansion* — also holds. That is, is it always the case that, if `t ==> t'` and `|- t' ∈ T`, then `|- t ∈ T`? If so, prove it. If not, give a counter-example. (You do not need to prove your counter-example in Coq, but feel free to do so.)

```
(* FILL IN HERE *)
```

□

Exercise: 2 stars (variation1)

Suppose, that we add this new rule to the typing relation:

```
      | T_SuccBool : ∀ t,
           |- t ∈ TBool →
           |- tsucc t ∈ TBool
```

Which of the following properties remain true in the presence of this rule? For each one, write either "remains true" or else "becomes false." If a property becomes false, give a counterexample.

- Determinism of `step`
- Progress
- Preservation

□

Exercise: 2 stars (variation2)

Suppose, instead, that we add this new rule to the `step` relation:

```
| ST_Funny1 : ∀ t2 t3,
    (tif ttrue t2 t3) ==> t3
```

Which of the above properties become false in the presence of this rule? For each one that does, give a counter-example.

□

Exercise: 2 stars, optional (variation3)

Suppose instead that we add this rule:

```
| ST_Funny2 : ∀ t1 t2 t2' t3,
    t2 ==> t2' →
    (tif t1 t2 t3) ==> (tif t1 t2' t3)
```

Which of the above properties become false in the presence of this rule? For each one that does, give a counter-example.

□

Exercise: 2 stars, optional (variation4)

Suppose instead that we add this rule:

```
| ST_Funny3 :
    (tpred tfalse) ==> (tpred (tpred tfalse))
```

Which of the above properties become false in the presence of this rule? For each one that does, give a counter-example.

□

Exercise: 2 stars, optional (variation5)

Suppose instead that we add this rule:

```
| T_Funny4 :
      |- tzero ∈ TBool
```

Which of the above properties become false in the presence of this rule? For each one that does, give a counter-example.

□

Exercise: 2 stars, optional (variation6)

Suppose instead that we add this rule:

```
      | T_Funny5 :
           |- tpred tzero ∈ TBool
```

Which of the above properties become false in the presence of this rule? For each one that does, give a counter-example.

□

Exercise: 3 stars, optional (more_variations)

Make up some exercises of your own along the same lines as the ones above. Try to find ways of selectively breaking properties — i.e., ways of changing the definitions that break just one of the properties and leave the others alone. □

Exercise: 1 star (remove_predzero)

The reduction rule `ST_PredZero` is a bit counter-intuitive: we might feel that it makes more sense for the predecessor of zero to be undefined, rather than being defined to be zero. Can we achieve this simply by removing the rule from the definition of `step`? Would doing so create any problems elsewhere?

```
(* FILL IN HERE *)
```

□

Exercise: 4 stars, advanced (prog_pres_bigstep)

Suppose our evaluation relation is defined in the big-step style. State appropriate analogs of the progress and preservation properties. (You do not need to prove them.)

Can you see any limitations of either of your properties? Do they allow for nonterminating commands? Why might we prefer the small-step semantics for stating preservation and progress?

```
(* FILL IN HERE *)
```

□