TYPECHECKING

A TYPECHECKER FOR STLC

The `has_type` relation of the STLC defines what it means for a term to belong to a type (in some context). But it doesn't, by itself, give us an algorithm for *checking* whether or not a term is well typed.

Fortunately, the rules defining `has_type` are *syntax directed* — that is, for every syntactic form of the language, there is just one rule that can be used to give a type to terms of that form. This makes it straightforward to translate the typing rules into clauses of a typechecking *function* that takes a term and a context and either returns the term's type or else signals that the term is not typable.

This short chapter constructs such a function and proves it correct.

```
Set Warnings "-notation-overridden,-parsing".
Require Import Coq.Bool.Bool.
Require Import Maps.
Require Import Smallstep.
Require Import Stlc.
Require MoreStlc.

Module STLCTypes.
Export STLC.
```

Comparing Types

First, we need a function to compare two types for equality...

```
Fixpoint beq_ty (T1 T2:ty) : bool :=
  match T1,T2 with
  | TBool, TBool ⇒
      true
  | TArrow T11 T12, TArrow T21 T22 ⇒
      andb (beq_ty T11 T21) (beq_ty T12 T22)
  | _,_ ⇒
```

```
      false
    end.
```

... and we need to establish the usual two-way connection between the boolean result returned by `beq_ty` and the logical proposition that its inputs are equal.

```
Lemma beq_ty_refl : ∀ T1,
  beq_ty T1 T1 = true.
+

Lemma beq_ty__eq : ∀ T1 T2,
  beq_ty T1 T2 = true → T1 = T2.
+
End STLCTypes.
```

The Typechecker

The typechecker works by walking over the structure of the given term, returning either `Some T` or `None`. Each time we make a recursive call to find out the types of the subterms, we need to pattern-match on the results to make sure that they are not `None`. Also, in the `tapp` case, we use pattern matching to extract the left- and right-hand sides of the function's arrow type (and fail if the type of the function is not `TArrow` T_{11} T_{12} for some T_{11} and T_{12}).

```
Module FirstTry.
Import STLCTypes.

Fixpoint type_check (Gamma:context) (t:tm) : option ty :=
  match t with
  | tvar x ⇒
      Gamma x
  | tabs x T11 t12 ⇒
      match type_check (update Gamma x T11) t12 with
      | Some T12 ⇒ Some (TArrow T11 T12)
      | _ ⇒ None
      end
  | tapp t1 t2 ⇒
      match type_check Gamma t1, type_check Gamma t2 with
      | Some (TArrow T11 T12),Some T2 ⇒
          if beq_ty T11 T2 then Some T12 else None
      | _,_ ⇒ None
      end
  | ttrue ⇒
      Some TBool
  | tfalse ⇒
      Some TBool
  | tif guard t f ⇒
      match type_check Gamma guard with
      | Some TBool ⇒
          match type_check Gamma t, type_check Gamma f with
```

```
          | Some T1, Some T2 ⇒
              if beq_ty T1 T2 then Some T1 else None
          | _,_ ⇒ None
          end
      | _ ⇒ None
      end
  end.

End FirstTry.
```

Digression: Improving the Notation

Before we consider the properties of this algorithm, let's write it out again in a cleaner way, using "monadic" notations in the style of Haskell to streamline the plumbing of options. First, we define a notation for composing two potentially failing (i.e., option-returning) computations:

```
Notation " x <- e1 ;; e2"
   := (match e1 with
         | Some x ⇒ e2
         | None ⇒ None
       end)
   (right associativity, at level 60).
```

Second, we define `return` and `fail` as synonyms for `Some` and `None`:

```
Notation " 'return' e "
  := (Some e) (at level 60).

Notation " 'fail' "
  := None.

Module STLCChecker.
Import STLCTypes.
```

Now we can write the same type-checking function in a more "imperative" style using these notations.

```
Fixpoint type_check (Gamma:context) (t:tm) : option ty :=
  match t with
  | tvar x ⇒
      match Gamma x with
      | Some T ⇒ return T
      | None ⇒ fail
      end
  | tabs x T11 t12 ⇒
      T12 <- type_check (update Gamma x T11) t12 ;;
      return (TArrow T11 T12)
  | tapp t1 t2 ⇒
      T1 <- type_check Gamma t1 ;;
      T2 <- type_check Gamma t2 ;;
      match T1 with
```

```
          | TArrow T11 T12 ⇒
            if beq_ty T11 T2 then return T12 else fail
          | _ ⇒ fail
          end
      | ttrue ⇒
          return TBool
      | tfalse ⇒
          return TBool
      | tif guard t1 t2 ⇒
          Tguard <- type_check Gamma guard ;;
          T1 <- type_check Gamma t1 ;;
          T2 <- type_check Gamma t2 ;;
          match Tguard with
          | TBool ⇒
            if beq_ty T1 T2 then return T1 else fail
          | _ ⇒ fail
          end
    end.
```

Properties

To verify that th typechecking algorithm is correct, we show that it is *sound* and *complete* for the original `has_type` relation — that is, `type_check` and `has_type` define the same partial function.

```
Theorem type_checking_sound : ∀ Gamma t T,
  type_check Gamma t = Some T → has_type Gamma t T.
+

Theorem type_checking_complete : ∀ Gamma t T,
  has_type Gamma t T → type_check Gamma t = Some T.
+

End STLCChecker.
```

Exercises

Exercise: 5 stars (typechecker_extensions)

In this exercise we'll extend the typechecker to deal with the extended features discussed in chapter MoreStlc. Your job is to fill in the omitted cases in the following.

```
Module TypecheckerExtensions.
Import MoreStlc.
Import STLCExtended.

Fixpoint beq_ty (T1 T2: ty) : bool :=
  match T1,T2 with
  | TNat, TNat ⇒
      true
```

```
    | TUnit, TUnit ⇒
      true
    | TArrow T11 T12, TArrow T21 T22 ⇒
      andb (beq_ty T11 T21) (beq_ty T12 T22)
    | TProd T11 T12, TProd T21 T22 ⇒
      andb (beq_ty T11 T21) (beq_ty T12 T22)
    | TSum T11 T12, TSum T21 T22 ⇒
      andb (beq_ty T11 T21) (beq_ty T12 T22)
    | TList T11, TList T21 ⇒
      beq_ty T11 T21
    | _,_ ⇒
      false
  end.

Lemma beq_ty_refl : ∀ T1,
  beq_ty T1 T1 = true.
Proof.
  intros T1.
  induction T1; simpl;
    try reflexivity;
    try (rewrite IHT1_1; rewrite IHT1_2; reflexivity);
    try (rewrite IHT1; reflexivity). Qed.

Lemma beq_ty__eq : ∀ T1 T2,
  beq_ty T1 T2 = true → T1 = T2.
Proof.
  intros T1.
  induction T1; intros T2 Hbeq; destruct T2; inversion Hbeq;
    try reflexivity;
    try (rewrite andb_true_iff in H0; inversion H0 as [Hbeq1
Hbeq2];
         apply IHT1_1 in Hbeq1; apply IHT1_2 in Hbeq2; subst;
auto);
    try (apply IHT1 in Hbeq; subst; auto).
 Qed.

Fixpoint type_check (Gamma:context) (t:tm) : option ty :=
  match t with
  | tvar x ⇒
      match Gamma x with
      | Some T ⇒ return T
      | None ⇒ fail
      end
  | tabs x T11 t12 ⇒
      T12 <- type_check (update Gamma x T11) t12 ;;
      return (TArrow T11 T12)
  | tapp t1 t2 ⇒
      T1 <- type_check Gamma t1 ;;
      T2 <- type_check Gamma t2 ;;
      match T1 with
      | TArrow T11 T12 ⇒
          if beq_ty T11 T2 then return T12 else fail
```

```
          | _ ⇒ fail
          end
      | tnat _ ⇒
          return TNat
      | tsucc t₁ ⇒
          T₁ <- type_check Gamma t₁ ;;
          match T₁ with
          | TNat ⇒ return TNat
          | _ ⇒ fail
          end
      | tpred t₁ ⇒
          T₁ <- type_check Gamma t₁ ;;
          match T₁ with
          | TNat ⇒ return TNat
          | _ ⇒ fail
          end
      | tmult t₁ t₂ ⇒
          T₁ <- type_check Gamma t₁ ;;
          T₂ <- type_check Gamma t₂ ;;
          match T₁, T₂ with
          | TNat, TNat ⇒ return TNat
          | _,_ ⇒ fail
          end
      | tif0 guard t f ⇒
          Tguard <- type_check Gamma guard ;;
          T₁ <- type_check Gamma t ;;
          T₂ <- type_check Gamma f ;;
          match Tguard with
          | TNat ⇒ if beq_ty T₁ T₂ then return T₁ else fail
          | _ ⇒ fail
          end
      (* FILL IN HERE *)
      | tlcase t₀ t₁ x₂₁ x₂₂ t₂ ⇒
          match type_check Gamma t₀ with
          | Some (TList T) ⇒
              match type_check Gamma t₁,
                    type_check (update (update Gamma x₂₂ (TList T))
  x₂₁ T) t₂ with
              | Some T₁', Some T₂' ⇒
                  if beq_ty T₁' T₂' then Some T₁' else None
              | _,_ ⇒ None
              end
          | _ ⇒ None
          end
      (* FILL IN HERE *)
      | _ ⇒ None (* ... and delete this line *)
      end.
```

Just for fun, we'll do the soundness proof with just a bit more automation than above, using these "mega-tactics":

```
Ltac invert_typecheck Gamma t T :=
  remember (type_check Gamma t) as TO;
  destruct TO as [T|];
  try solve_by_invert; try (inversion H0; eauto); try (subst;
eauto).

Ltac analyze T T1 T2 :=
  destruct T as [T1 T2| | | T1 T2| T1 T2| T1]; try
solve_by_invert.

Ltac fully_invert_typecheck Gamma t T T1 T2 :=
  let TX := fresh T in
  remember (type_check Gamma t) as TO;
  destruct TO as [TX|]; try solve_by_invert;
  destruct TX as [T1 T2| | | T1 T2| T1 T2| T1];
  try solve_by_invert; try (inversion H0; eauto); try (subst;
eauto).

Ltac case_equality S T :=
  destruct (beq_ty S T) eqn: Heqb;
  inversion H0; apply beq_ty__eq in Heqb; subst; subst; eauto.

Theorem type_checking_sound : ∀ Gamma t T,
  type_check Gamma t = Some T → has_type Gamma t T.
Proof with eauto.
  intros Gamma t. generalize dependent Gamma.
  induction t; intros Gamma T Htc; inversion Htc.
  - (* tvar *) rename s into x. destruct (Gamma x) eqn:H.
    rename t into T'. inversion H0. subst. eauto.
solve_by_invert.
  - (* tapp *)
    invert_typecheck Gamma t1 T1.
    invert_typecheck Gamma t2 T2.
    analyze T1 T11 T12.
    case_equality T11 T2.
  - (* tabs *)
    rename s into x. rename t into T1.
    remember (update Gamma x T1) as Gamma'.
    invert_typecheck Gamma' t0 T0.
  - (* tnat *) eauto.
  - (* tsucc *)
    rename t into t1.
    fully_invert_typecheck Gamma t1 T1 T11 T12.
  - (* tpred *)
    rename t into t1.
    fully_invert_typecheck Gamma t1 T1 T11 T12.
  - (* tmult *)
    invert_typecheck Gamma t1 T1.
    invert_typecheck Gamma t2 T2.
    analyze T1 T11 T12; analyze T2 T21 T22.
    inversion H0. subst. eauto.
  - (* tif0 *)
```

```
      invert_typecheck Gamma t1 T1.
      invert_typecheck Gamma t2 T2.
      invert_typecheck Gamma t3 T3.
      destruct T1; try solve_by_invert.
      case_equality T2 T3.
    (* FILL IN HERE *)
    - (* tlcase *)
      rename s into x31. rename s0 into x32.
      fully_invert_typecheck Gamma t1 T1 T11 T12.
      invert_typecheck Gamma t2 T2.
      remember (update (update Gamma x32 (TList T11)) x31 T11) as
  Gamma'2.
      invert_typecheck Gamma'2 t3 T3.
      case_equality T2 T3.
    (* FILL IN HERE *)
  Qed.

  Theorem type_checking_complete : ∀ Gamma t T,
    has_type Gamma t T → type_check Gamma t = Some T.
  Proof.
    intros Gamma t T Hty.
    induction Hty; simpl;
      try (rewrite IHHty);
      try (rewrite IHHty1);
      try (rewrite IHHty2);
      try (rewrite IHHty3);
      try (rewrite (beq_ty_refl T));
      try (rewrite (beq_ty_refl T1));
      try (rewrite (beq_ty_refl T2));
      eauto.
    - destruct (Gamma x); try solve_by_invert. eauto.
    Admitted. (* ... and delete this line *)
  (*
  Qed. (* ... and uncomment this one *)
  *)
  End TypecheckerExtensions.
```

□

Exercise: 5 stars, optional (stlc_step_function)

Above, we showed how to write a typechecking function and prove it sound and complete for the typing relation. Do the same for the operational semantics — i.e., write a function `stepf` of type `tm → option tm` and prove that it is sound and complete with respect to `step` from chapter MoreStlc.

```
  Module StepFunction.
  Import TypecheckerExtensions.

  (* FILL IN HERE *)
  End StepFunction.
```

□

Exercise: 5 stars, optional (stlc_impl)

Using the Imp parser described in the ImpParser chapter of *Logical Foundations* as a guide, build a parser for extended Stlc programs. Combine it with the typechecking and stepping functions from the above exercises to yield a complete typechecker and interpreter for this language.

```
Module StlcImpl.
Import StepFunction.

(* FILL IN HERE *)
End StlcImpl.
```

□