RECORDS

ADDING RECORDS TO STLC

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

Adding Records

We saw in chapter MoreStlc how records can be treated as just syntactic sugar for nested uses of products. This is OK for simple examples, but the encoding is informal (in reality, if we actually treated records this way, it would be carried out in the parser, which we are eliding here), and anyway it is not very efficient. So it is also interesting to see how records can be treated as first-class citizens of the language. This chapter shows how.

Recall the informal definitions we gave before:

Syntax:

```
t ::=                             Terms:
    | {i1=t1, ..., in=tn}           record
    | t.i                           projection
    | ...

v ::=                             Values:
    | {i1=v1, ..., in=vn}           record value
    | ...

T ::=                             Types:
    | {i1:T1, ..., in:Tn}           record type
    | ...
```

Reduction:

```
                        ti ==> ti'
---------------------------------------------------------------- (ST_Rcd)
{i1=v1, ..., im=vm, in=tn, ...} ==> {i1=v1, ..., im=vm, in=tn', ...}
```

```
 t1 ==> t1'
------------------  (ST_Proj1)
t1.i ==> t1'.i
```

```
---------------------------  (ST_ProjRcd)
{..., i=vi, ...}.i ==> vi
```

Typing:

```
  Gamma |- t1 : T1     ...     Gamma |- tn : Tn
--------------------------------------------------  (T_Rcd)
Gamma |- {i1=t1, ..., in=tn} : {i1:T1, ..., in:Tn}

Gamma |- t : {..., i:Ti, ...}
------------------------------  (T_Proj)
     Gamma |- t.i : Ti
```

Formalizing Records

```
Module STLCExtendedRecords.
```

Syntax and Operational Semantics

The most obvious way to formalize the syntax of record types would be this:

```
Module FirstTry.

Definition alist (X : Type) := list (string * X).

Inductive ty : Type :=
  | TBase : string → ty
  | TArrow : ty → ty → ty
  | TRcd : (alist ty) → ty.
```

Unfortunately, we encounter here a limitation in Coq: this type does not automatically give us the induction principle we expect: the induction hypothesis in the `TRcd` case doesn't give us any information about the `ty` elements of the list, making it useless for the proofs we want to do.

```
(* Check ty_ind.
   ====>
    ty_ind :
      forall P : ty -> Prop,
        (forall i : id, P (TBase i)) ->
        (forall t : ty, P t -> forall t0 : ty, P t0
                            -> P (TArrow t t0)) ->
        (forall a : alist ty, P (TRcd a)) ->    (* ??? *)
        forall t : ty, P t
*)

End FirstTry.
```

It is possible to get a better induction principle out of Coq, but the details of how this is done are not very pretty, and the principle we obtain is not as intuitive to use as the ones Coq generates automatically for simple `Inductive` definitions.

Fortunately, there is a different way of formalizing records that is, in some ways, even simpler and more natural: instead of using the standard Coq `list` type, we can essentially incorporate its constructors ("nil" and "cons") in the syntax of our types.

```
Inductive ty : Type :=
  | TBase : string → ty
  | TArrow : ty → ty → ty
  | TRNil : ty
  | TRCons : string → ty → ty → ty.
```

Similarly, at the level of terms, we have constructors `trnil`, for the empty record, and `trcons`, which adds a single field to the front of a list of fields.

```
Inductive tm : Type :=
  | tvar : string → tm
  | tapp : tm → tm → tm
  | tabs : string → ty → tm → tm
  (* records *)
  | tproj : tm → string → tm
  | trnil : tm
  | trcons : string → tm → tm → tm.
```

Some examples...

```
Open Scope string_scope.

Notation a := "a".
Notation f := "f".
Notation g := "g".
Notation l := "l".
Notation A := (TBase "A").
Notation B := (TBase "B").
Notation k := "k".
Notation i1 := "i1".
Notation i2 := "i2".
```

{ i_1:A }

```
(* Check (TRCons i1 A TRNil). *)
```

{ i_1:A→B, i_2:A }

```
(* Check (TRCons i1 (TArrow A B)
           (TRCons i2 A TRNil)). *)
```

Well-Formedness

One issue with generalizing the abstract syntax for records from lists to the nil/cons presentation is that it introduces the possibility of writing strange types like this...

```
Definition weird_type := TRCons X A B.
```

where the "tail" of a record type is not actually a record type!

We'll structure our typing judgement so that no ill-formed types like `weird_type` are ever assigned to terms. To support this, we define predicates `record_ty` and `record_tm`, which identify record types and terms, and `well_formed_ty` which rules out the ill-formed types.

First, a type is a record type if it is built with just `TRNil` and `TRCons` at the outermost level.

```
Inductive record_ty : ty → Prop :=
  | RTnil :
        record_ty TRNil
  | RTcons : ∀ i T1 T2,
        record_ty (TRCons i T1 T2).
```

With this, we can define well-formed types.

```
Inductive well_formed_ty : ty → Prop :=
  | wfTBase : ∀ i,
        well_formed_ty (TBase i)
  | wfTArrow : ∀ T1 T2,
        well_formed_ty T1 →
        well_formed_ty T2 →
        well_formed_ty (TArrow T1 T2)
  | wfTRNil :
        well_formed_ty TRNil
  | wfTRCons : ∀ i T1 T2,
        well_formed_ty T1 →
        well_formed_ty T2 →
        record_ty T2 →
        well_formed_ty (TRCons i T1 T2).

Hint Constructors record_ty well_formed_ty.
```

Note that `record_ty` is not recursive — it just checks the outermost constructor. The `well_formed_ty` property, on the other hand, verifies that the whole type is well formed in the sense that the tail of every record (the second argument to `TRCons`) is a record.

Of course, we should also be concerned about ill-formed terms, not just types; but typechecking can rules those out without the help of an extra `well_formed_tm` definition because it already examines the structure of terms. All we need is an analog of `record_ty` saying that a term is a record term if it is built with `trnil` and `trcons`.

```
Inductive record_tm : tm → Prop :=
  | rtnil :
        record_tm trnil
  | rtcons : ∀ i t1 t2,
        record_tm (trcons i t1 t2).

Hint Constructors record_tm.
```

Substitution

Substitution extends easily.

```
Fixpoint subst (x:string) (s:tm) (t:tm) : tm :=
  match t with
  | tvar y ⇒ if beq_string x y then s else t
  | tabs y T t1 ⇒ tabs y T
```

```
                      (if beq_string x y then t1 else (subst x s t1))
    | tapp t1 t2 ⇒ tapp (subst x s t1) (subst x s t2)
    | tproj t1 i ⇒ tproj (subst x s t1) i
    | trnil ⇒ trnil
    | trcons i t1 tr1 ⇒ trcons i (subst x s t1) (subst x s tr1)
  end.

Notation "'[' x ':=' s ']' t" := (subst x s t) (at level 20).
```

Reduction

A record is a value if all of its fields are.

```
Inductive value : tm → Prop :=
  | v_abs : ∀ x T11 t12,
      value (tabs x T11 t12)
  | v_rnil : value trnil
  | v_rcons : ∀ i v1 vr,
      value v1 →
      value vr →
      value (trcons i v1 vr).

Hint Constructors value.
```

To define reduction, we'll need a utility function for extracting one field from record term:

```
Fixpoint tlookup (i:string) (tr:tm) : option tm :=
  match tr with
  | trcons i' t tr' ⇒ if beq_string i i' then Some t else tlookup i
tr'
  | _ ⇒ None
  end.
```

The `step` function uses this term-level lookup function in the projection rule.

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

Inductive step : tm → tm → Prop :=
  | ST_AppAbs : ∀ x T11 t12 v2,
         value v2 →
         (tapp (tabs x T11 t12) v2) ==> ([x:=v2]t12)
  | ST_App1 : ∀ t1 t1' t2,
         t1 ==> t1' →
         (tapp t1 t2) ==> (tapp t1' t2)
  | ST_App2 : ∀ v1 t2 t2',
         value v1 →
         t2 ==> t2' →
         (tapp v1 t2) ==> (tapp v1 t2')
  | ST_Proj1 : ∀ t1 t1' i,
        t1 ==> t1' →
        (tproj t1 i) ==> (tproj t1' i)
  | ST_ProjRcd : ∀ tr i vi,
        value tr →
        tlookup i tr = Some vi →
        (tproj tr i) ==> vi
```

```
    | ST_Rcd_Head : ∀ i t1 t1' tr2,
          t1 ==> t1' →
          (trcons i t1 tr2) ==> (trcons i t1' tr2)
    | ST_Rcd_Tail : ∀ i v1 tr2 tr2',
          value v1 →
          tr2 ==> tr2' →
          (trcons i v1 tr2) ==> (trcons i v1 tr2')

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

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

Hint Constructors step.
```

Typing

Next we define the typing rules. These are nearly direct transcriptions of the inference rules shown above: the only significant difference is the use of `well_formed_ty`. In the informal presentation we used a grammar that only allowed well-formed record types, so we didn't have to add a separate check.

One sanity condition that we'd like to maintain is that, whenever `has_type Gamma t T` holds, will also be the case that `well_formed_ty T`, so that `has_type` never assigns ill-formed types to terms. In fact, we prove this theorem below. However, we don't want to clutter the definition of `has_type` with unnecessary uses of `well_formed_ty`. Instead, we place `well_formed_ty` checks only where needed: where an inductive call to `has_type` won't already be checking the well-formedness of a type. For example, we check `well_formed_ty T` in the `T_Var` case, because there is no inductive `has_type` call that would enforce this. Similarly, in the `T_Abs` case, we require a proof of `well_formed_ty` T_{11} because the inductive call to `has_type` only guarantees that T_{12} is well-formed.

```
Fixpoint Tlookup (i:string) (Tr:ty) : option ty :=
  match Tr with
  | TRCons i' T Tr' ⇒
     if beq_string i i' then Some T else Tlookup i Tr'
  | _ ⇒ None
  end.

Definition context := partial_map ty.

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

Inductive has_type : context → tm → ty → Prop :=
  | T_Var : ∀ Gamma x T,
     Gamma x = Some T →
     well_formed_ty T →
     Gamma |- (tvar x) ∈ T
  | T_Abs : ∀ Gamma x T11 T12 t12,
     well_formed_ty T11 →
     (update Gamma x T11) |- t12 ∈ T12 →
     Gamma |- (tabs x T11 t12) ∈ (TArrow T11 T12)
  | T_App : ∀ T1 T2 Gamma t1 t2,
```

```
        Gamma |- t₁ ∈ (TArrow T₁ T₂) →
        Gamma |- t₂ ∈ T₁ →
        Gamma |- (tapp t₁ t₂) ∈ T₂
    (* records: *)
    | T_Proj : ∀ Gamma i t Ti Tr,
        Gamma |- t ∈ Tr →
        Tlookup i Tr = Some Ti →
        Gamma |- (tproj t i) ∈ Ti
    | T_RNil : ∀ Gamma,
        Gamma |- trnil ∈ TRNil
    | T_RCons : ∀ Gamma i t T tr Tr,
        Gamma |- t ∈ T →
        Gamma |- tr ∈ Tr →
        record_ty Tr →
        record_tm tr →
        Gamma |- (trcons i t tr) ∈ (TRCons i T Tr)

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

Hint Constructors has_type.
```

Examples

Exercise: 2 stars (examples)

Finish the proofs below. Feel free to use Coq's automation features in this proof. However, if you are not confident about how the type system works, you may want to carry out the proofs first using the basic features (`apply` instead of `eapply`, in particular) and then perhaps compress it using automation. Before starting to prove anything, make sure you understand what it is saying.

```
(* GRADE_THEOREM 0.5: typing_example_2 *)
Lemma typing_example_2 :
  empty |-
    (tapp (tabs a (TRCons i₁ (TArrow A A)
                      (TRCons i₂ (TArrow B B)
                       TRNil))
              (tproj (tvar a) i₂))
            (trcons i₁ (tabs a A (tvar a))
            (trcons i₂ (tabs a B (tvar a))
             trnil))) ∈
    (TArrow B B).
Proof.
  (* FILL IN HERE *) Admitted.

(* GRADE_THEOREM 0.5: typing_nonexample *)
Example typing_nonexample :
  ¬ ∃ T,
      (update empty a (TRCons i₂ (TArrow A A)
                                TRNil)) |-
                (trcons i₁ (tabs a B (tvar a)) (tvar a)) ∈
                T.
Proof.
  (* FILL IN HERE *) Admitted.

Example typing_nonexample_2 : ∀ y,
  ¬ ∃ T,
```

```
      (update empty y A) |-
             (tapp (tabs a (TRCons i₁ A TRNil)
                       (tproj (tvar a) i₁))
                     (trcons i₁ (tvar y) (trcons i₂ (tvar y) trnil))) ∈
             T.
Proof.
  (* FILL IN HERE *) Admitted.
```

Properties of Typing

The proofs of progress and preservation for this system are essentially the same as for the pure simply typed lambda-calculus, but we need to add some technical lemmas involving records.

Well-Formedness

```
Lemma wf_rcd_lookup : ∀ i T Ti,
  well_formed_ty T →
  Tlookup i T = Some Ti →
  well_formed_ty Ti.
+

Lemma step_preserves_record_tm : ∀ tr tr',
  record_tm tr →
  tr ==> tr' →
  record_tm tr'.
+

Lemma has_type__wf : ∀ Gamma t T,
  Gamma |- t ∈ T → well_formed_ty T.
+
```

Field Lookup

Lemma: If `empty |- v : T` and `Tlookup i T` returns `Some Ti`, then `tlookup i v` returns `Some ti` for some term `ti` such that `empty |- ti ∈ Ti`.

Proof: By induction on the typing derivation `Htyp`. Since `Tlookup i T = Some Ti`, `T` must be a record type, this and the fact that `v` is a value eliminate most cases by inspection, leaving only the `T_RCons` case.

If the last step in the typing derivation is by `T_RCons`, then `t = trcons` i_0 `t tr` and `T = TRCons` i_0 `T Tr` for some i_0, `t`, `tr`, `T` and `Tr`.

This leaves two possiblities to consider - either i_0 = `i` or not.

- If `i` = i_0, then since `Tlookup i (TRCons` i_0 `T Tr) = Some Ti` we have `T = Ti`. It follows that `t` itself satisfies the theorem.
- On the other hand, suppose `i` ≠ i_0. Then

  `Tlookup i T = Tlookup i Tr`

  and

```
tlookup i t = tlookup i tr,
```

so the result follows from the induction hypothesis. □

Here is the formal statement:

```
Lemma lookup_field_in_value : ∀ v T i Ti,
  value v →
  empty |- v ∈ T →
  Tlookup i T = Some Ti →
  ∃ ti, tlookup i v = Some ti ∧ empty |- ti ∈ Ti.
```

+

Progress

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

+

Context Invariance

```
Inductive appears_free_in : string → tm → Prop :=
  | afi_var : ∀ x,
      appears_free_in x (tvar x)
  | afi_app1 : ∀ x t1 t2,
      appears_free_in x t1 → appears_free_in x (tapp t1 t2)
  | afi_app2 : ∀ x t1 t2,
      appears_free_in x t2 → appears_free_in x (tapp t1 t2)
  | afi_abs : ∀ x y T11 t12,
        y ≠ x →
        appears_free_in x t12 →
        appears_free_in x (tabs y T11 t12)
  | afi_proj : ∀ x t i,
     appears_free_in x t →
     appears_free_in x (tproj t i)
  | afi_rhead : ∀ x i ti tr,
      appears_free_in x ti →
      appears_free_in x (trcons i ti tr)
  | afi_rtail : ∀ x i ti tr,
      appears_free_in x tr →
      appears_free_in x (trcons i ti tr).

Hint Constructors appears_free_in.

Lemma context_invariance : ∀ Gamma Gamma' t S,
     Gamma |- t ∈ S →
     (∀ x, appears_free_in x t → Gamma x = Gamma' x) →
     Gamma' |- t ∈ S.
```

+

```
Lemma free_in_context : ∀ x t T Gamma,
   appears_free_in x t →
   Gamma |- t ∈ T →
   ∃ T', Gamma x = Some T'.
```

+

Preservation

```
Lemma substitution_preserves_typing : ∀ Gamma x U v t S,
     (update Gamma x U) |- t ∈ S →
     empty |- v ∈ U →
     Gamma |- ([x:=v]t) ∈ S.
Proof with eauto.
  (* Theorem: If Gamma,x:U |- t : S and empty |- v : U, then
     Gamma |- (x:=vt) S. *)
  intros Gamma x U v t S Htypt Htypv.
  generalize dependent Gamma. generalize dependent S.
  (* Proof: By induction on the term t.  Most cases follow
     directly from the IH, with the exception of tvar,
     tabs, trcons. The former aren't automatic because we
     must reason about how the variables interact. In the
     case of trcons, we must do a little extra work to show
     that substituting into a term doesn't change whether
     it is a record term. *)
  induction t;
    intros S Gamma Htypt; simpl; inversion Htypt; subst...
  - (* tvar *)
    simpl. rename s into y.
    (* If t = y, we know that
         empty |- v : U and
         Gamma,x:U |- y : S
       and, by inversion, update Gamma x U y = Some S.
       We want to show that Gamma |- [x:=v]y : S.

       There are two cases to consider: either x=y or x≠y. *)
    unfold update, t_update in H0.
    destruct (beq_stringP x y) as [Hxy|Hxy].
    + (* x=y *)
    (* If x = y, then we know that U = S, and that
       [x:=v]y = v. So what we really must show is that
       if empty |- v : U then Gamma |- v : U.  We have
        already proven a more general version of this theorem,
        called context invariance! *)
      subst.
      inversion H0; subst. clear H0.
      eapply context_invariance...
      intros x Hcontra.
      destruct (free_in_context _ _ S empty Hcontra)
        as [T' HT']...
      inversion HT'.
    + (* x<>y *)
    (* If x ≠ y, then Gamma y = Some S and the substitution
       has no effect.  We can show that Gamma |- y : S by
       T_Var. *)
      apply T_Var...
  - (* tabs *)
    rename s into y. rename t into T11.
    (* If t = tabs y T11 t0, then we know that
         Gamma,x:U |- tabs y T11 t0 : T11→T12
         Gamma,x:U,y:T11 |- t0 : T12
         empty |- v : U
```

```
        As our IH, we know that forall S Gamma,
          Gamma,x:U |- t0 : S → Gamma |- [x:=v]t0 S.

        We can calculate that
         [x:=v]t = tabs y T11 (if beq_string x y then t0 else
[x:=v]t0) ,
        and we must show that Gamma |- [x:=v]t : T11→T12.  We know
        we will do so using T_Abs, so it remains to be shown that:
          Gamma,y:T11 |- if beq_string x y then t0 else [x:=v]t0 : T12
        We consider two cases: x = y and x ≠ y. *)
    apply T_Abs...
    destruct (beq_stringP x y) as [Hxy|Hxy].
    + (* x=y *)
      (* If x = y, then the substitution has no effect.  Context
         invariance shows that Gamma,y:U,y:T11 and Gamma,y:T11 are
         equivalent.  Since t0 : T12 under the former context,
         this is also the case under the latter. *)
      eapply context_invariance...
      subst.
      intros x Hafi. unfold update, t_update.
      destruct (beq_string y x)...
    + (* x<>y *)
      (* If x ≠ y, then the IH and context invariance allow
         us to show that
           Gamma,x:U,y:T11 |- t0 : T12       =>
           Gamma,y:T11,x:U |- t0 : T12       =>
           Gamma,y:T11 |- [x:=v]t0 : T12 *)
      apply IHt. eapply context_invariance...
      intros z Hafi. unfold update, t_update.
      destruct (beq_stringP y z)...
      subst. rewrite false_beq_string...
  - (* trcons *)
    apply T_RCons... inversion H7; subst; simpl...
Qed.

Theorem preservation : ∀ t t' T,
     empty |- t ∈ T →
     t ==> t' →
     empty |- t' ∈ T.
Proof with eauto.
  intros t t' T HT.
  (* Theorem: If empty |- t : T and t ==> t', then
     empty |- t' : T. *)
  remember (@empty ty) as Gamma. generalize dependent HeqGamma.
  generalize dependent t'.
  (* Proof: By induction on the given typing derivation.
     Many cases are contradictory (T_Var, T_Abs) or follow
     directly from the IH (T_RCons).  We show just the
     interesting ones. *)
  induction HT;
    intros t' HeqGamma HE; subst; inversion HE; subst...
  - (* T_App *)
    (* If the last rule used was T_App, then t = t1 t2,
       and three rules could have been used to show t ==> t':
       ST_App1, ST_App2, and ST_AppAbs. In the first two
       cases, the result follows directly from the IH. *)
    inversion HE; subst...
```

```
      + (* ST_AppAbs *)
        (* For the third case, suppose
             t₁ = tabs x T₁₁ t₁₂
           and
             t₂ = v₂.  We must show that empty |- [x:=v₂]t₁₂ : T₂.
           We know by assumption that
               empty |- tabs x T₁₁ t₁₂ : T₁→T₂
           and by inversion
               x:T₁ |- t₁₂ : T₂
           We have already proven that substitution_preserves_typing and
               empty |- v₂ : T₁
           by assumption, so we are done. *)
        apply substitution_preserves_typing with T₁...
        inversion HT₁...
    - (* T_Proj *)
      (* If the last rule was T_Proj, then t = tproj t₁ i.
         Two rules could have caused t ==> t': T_Proj1 and
         T_ProjRcd.  The typing of t' follows from the IH
         in the former case, so we only consider T_ProjRcd.

         Here we have that t is a record value.  Since rule
         T_Proj was used, we know empty |- t ∈ Tr and
         Tlookup i Tr = Some Ti for some i and Tr.
         We may therefore apply lemma lookup_field_in_value
         to find the record element this projection steps to. *)
      destruct (lookup_field_in_value _ _ _ _ H₂ HT H)
        as [vi [Hget Htyp]].
      rewrite H₄ in Hget. inversion Hget. subst...
    - (* T_RCons *)
      (* If the last rule was T_RCons, then t = trcons i t tr
         for some i, t and tr such that record_tm tr.  If
         the step is by ST_Rcd_Head, the result is immediate by
         the IH.  If the step is by ST_Rcd_Tail, tr ==> tr₂'
         for some tr₂' and we must also use lemma step_preserves_record_tm
         to show record_tm tr₂'. *)
      apply T_RCons... eapply step_preserves_record_tm...
  Qed.
```
□

```
  End STLCExtendedRecords.
```