SOFTWARE FOUNDATIONS

VOLUME 2: PROGRAMMING LANGUAGE FOUNDATIONS

RECORDSUB

SUBTYPING WITH RECORDS

In this chapter, we combine two significant extensions of the pure STLC — records (from chapter Records) and subtyping (from chapter Sub) — and explore their interactions. Most of the concepts have already been discussed in those chapters, so the presentation here is somewhat terse. We just comment where things are nonstandard.

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

Core Definitions

Syntax

```
Inductive ty : Type :=
  (* proper types *)
  | TTop : ty
  | TBase : string → ty
  | TArrow : ty → ty → ty
  (* record types *)
  | TRNil : ty
  | TRCons : string → ty → ty → ty.

Inductive tm : Type :=
  (* proper terms *)
  | tvar : string → tm
  | tapp : tm → tm → tm
  | tabs : string → ty → tm → tm
  | tproj : tm → string → tm
  (* record terms *)
```

```
  | trnil : tm
  | trcons : string → tm → tm → tm.
```

Well-Formedness

The syntax of terms and types is a bit too loose, in the sense that it admits things like a record type whose final "tail" is `Top` or some arrow type rather than `Nil`. To avoid such cases, it is useful to assume that all the record types and terms that we see will obey some simple well-formedness conditions.

An interesting technical question is whether the basic properties of the system -- progress and preservation -- remain true if we drop these conditions. I believe they do, and I would encourage motivated readers to try to check this by dropping the conditions from the definitions of typing and subtyping and adjusting the proofs in the rest of the chapter accordingly. This is not a trivial exercise (or I'd have done it!), but it should not involve changing the basic structure of the proofs. If someone does do it, please let me know. --BCP 5/16.

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

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

Inductive well_formed_ty : ty → Prop :=
  | wfTTop :
        well_formed_ty TTop
  | 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 record_tm well_formed_ty.
```

Substitution

Substitution and reduction are as before.

```
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 tr2 ⇒ trcons i (subst x s t1) (subst x s tr2)
  end.

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

Reduction

```
Inductive value : tm → Prop :=
  | v_abs : ∀ x T t,
      value (tabs x T t)
  | v_rnil : value trnil
  | v_rcons : ∀ i v vr,
      value v →
      value vr →
      value (trcons i v vr).

Hint Constructors value.

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.

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.

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

Inductive step : tm → tm → Prop :=
  | ST_AppAbs : ∀ x T t12 v2,
         value v2 →
         (tapp (tabs x T 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 : ∀ tr tr' i,
        tr ==> tr' →
        (tproj tr i) ==> (tproj tr' 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).

Hint Constructors step.
```

Subtyping

Now we come to the interesting part, where the features we've added start to interact. We begin by defining the subtyping relation and developing some of its important technical properties.

Definition

The definition of subtyping is essentially just what we sketched in the discussion of record subtyping in chapter Sub, but we need to add well-formedness side conditions to some of the rules. Also, we replace the "n-ary" width, depth, and permutation subtyping rules by binary rules that deal with just the first field.

```
Reserved Notation "T '<:' U" (at level 40).

Inductive subtype : ty → ty → Prop :=
  (* Subtyping between proper types *)
  | S_Refl : ∀ T,
    well_formed_ty T →
    T <: T
  | S_Trans : ∀ S U T,
    S <: U →
    U <: T →
    S <: T
  | S_Top : ∀ S,
    well_formed_ty S →
    S <: TTop
  | S_Arrow : ∀ S1 S2 T1 T2,
```

```
    T₁ <: S₁ →
    S₂ <: T₂ →
    TArrow S₁ S₂ <: TArrow T₁ T₂
  (* Subtyping between record types *)
  | S_RcdWidth : ∀ i T₁ T₂,
    well_formed_ty (TRCons i T₁ T₂) →
    TRCons i T₁ T₂ <: TRNil
  | S_RcdDepth : ∀ i S₁ T₁ Sr₂ Tr₂,
    S₁ <: T₁ →
    Sr₂ <: Tr₂ →
    record_ty Sr₂ →
    record_ty Tr₂ →
    TRCons i S₁ Sr₂ <: TRCons i T₁ Tr₂
  | S_RcdPerm : ∀ i₁ i₂ T₁ T₂ Tr₃,
    well_formed_ty (TRCons i₁ T₁ (TRCons i₂ T₂ Tr₃)) →
    i₁ ≠ i₂ →
       TRCons i₁ T₁ (TRCons i₂ T₂ Tr₃)
    <: TRCons i₂ T₂ (TRCons i₁ T₁ Tr₃)

where "T '<:' U" := (subtype T U).

Hint Constructors subtype.
```

Examples

```
Module Examples.
Open Scope string_scope.

Notation x := "x".
Notation y := "y".
Notation z := "z".
Notation j := "j".
Notation k := "k".
Notation i := "i".
Notation A := (TBase "A").
Notation B := (TBase "B").
Notation C := (TBase "C").

Definition TRcd_j :=
  (TRCons j (TArrow B B) TRNil). (* {j:B->B} *)
Definition TRcd_kj :=
  TRCons k (TArrow A A) TRcd_j. (* {k:C->C,j:B->B} *)

Example subtyping_example_0 :
  subtype (TArrow C TRcd_kj)
          (TArrow C TRNil).
(* C->{k:A->A,j:B->B} <: C->{} *)
Proof.
  apply S_Arrow.
    apply S_Refl. auto.
    unfold TRcd_kj, TRcd_j. apply S_RcdWidth; auto.
Qed.
```

The following facts are mostly easy to prove in Coq. To get full benefit, make sure you also understand how to prove them on paper!

Exercise: 2 stars (subtyping_example_1)

```
Example subtyping_example_1 :
  subtype TRcd_kj TRcd_j.
(* {k:A->A,j:B->B} <: {j:B->B} *)
Proof with eauto.
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 1 star (subtyping_example_2)

```
Example subtyping_example_2 :
  subtype (TArrow TTop TRcd_kj)
          (TArrow (TArrow C C) TRcd_j).
(* Top->{k:A->A,j:B->B} <: (C->C)->{j:B->B} *)
Proof with eauto.
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 1 star (subtyping_example_3)

```
Example subtyping_example_3 :
  subtype (TArrow TRNil (TRCons j A TRNil))
          (TArrow (TRCons k B TRNil) TRNil).
(* {}->{j:A} <: {k:B}->{} *)
Proof with eauto.
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 2 stars (subtyping_example_4)

```
Example subtyping_example_4 :
  subtype (TRCons x A (TRCons y B (TRCons z C TRNil)))
          (TRCons z C (TRCons y B (TRCons x A TRNil))).
(* {x:A,y:B,z:C} <: {z:C,y:B,x:A} *)
Proof with eauto.
  (* FILL IN HERE *) Admitted.
```

□

```
End Examples.
```

Properties of Subtyping

Well-Formedness

To get started proving things about subtyping, we need a couple of technical lemmas that intuitively (1) allow us to extract the well-formedness assumptions embedded in subtyping derivations and (2) record the fact that fields of well-formed record types are themselves well-formed types.

```
Lemma subtype__wf : ∀ S T,
  subtype S T →
```

```
  well_formed_ty T ∧ well_formed_ty S.
```

+

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

+

Field Lookup

The record matching lemmas get a little more complicated in the presence of subtyping, for two reasons. First, record types no longer necessarily describe the exact structure of the corresponding terms. And second, reasoning by induction on typing derivations becomes harder in general, because typing is no longer syntax directed.

```
Lemma rcd_types_match : ∀ S T i Ti,
  subtype S T →
  Tlookup i T = Some Ti →
  ∃ Si, Tlookup i S = Some Si ∧ subtype Si Ti.
```

+

Exercise: 3 stars (rcd_types_match_informal)

Write a careful informal proof of the `rcd_types_match` lemma.

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

☐

Inversion Lemmas

Exercise: 3 stars, optional (sub_inversion_arrow)

```
Lemma sub_inversion_arrow : ∀ U V1 V2,
     subtype U (TArrow V1 V2) →
     ∃ U1, ∃ U2,
       (U=(TArrow U1 U2)) ∧ (subtype V1 U1) ∧ (subtype U2 V2).
```

+

☐

Typing

```
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
  | T_Proj : ∀ Gamma i t T Ti,
      Gamma |- t ∈ T →
      Tlookup i T = Some Ti →
      Gamma |- tproj t i ∈ Ti
  (* Subsumption *)
  | T_Sub : ∀ Gamma t S T,
      Gamma |- t ∈ S →
      subtype S T →
      Gamma |- t ∈ T
  (* Rules for record terms *)
  | 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.
```

Typing Examples

```
Module Examples2.
Import Examples.
```

Exercise: 1 star (typing_example_0)

```
Definition trcd_kj :=
  (trcons k (tabs z A (tvar z))
          (trcons j (tabs z B (tvar z))
                    trnil)).

Example typing_example_0 :
  has_type empty
          (trcons k (tabs z A (tvar z))
                    (trcons j (tabs z B (tvar z))
                              trnil))
          TRcd_kj.
(* empty |- {k=(\z:A.z), j=(\z:B.z)} : {k:A->A,j:B->B} *)
+
□
```

Exercise: 2 stars (typing_example_1)

```
Example typing_example_1 :
  has_type empty
           (tapp (tabs x TRcd_j (tproj (tvar x) j))
                   (trcd_kj))
           (TArrow B B).
(* empty |- (\x:{k:A->A,j:B->B}. x.j)
              {k=(\z:A.z), j=(\z:B.z)}
         : B->B *)
```

+

□

Exercise: 2 stars, optional (typing_example_2)

```
Example typing_example_2 :
  has_type empty
           (tapp (tabs z (TArrow (TArrow C C) TRcd_j)
                           (tproj (tapp (tvar z)
                                         (tabs x C (tvar x)))
                                   j))
                   (tabs z (TArrow C C) trcd_kj))
           (TArrow B B).
(* empty |- (\z:(C->C)->{j:B->B}. (z (\x:C.x)).j)
              (\z:C->C. {k=(\z:A.z), j=(\z:B.z)})
           : B->B *)
```

+

□

```
End Examples2.
```

Properties of Typing

Well-Formedness

```
Lemma has_type__wf : ∀ Gamma t T,
  has_type Gamma t T → well_formed_ty T.
```

+

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

+

Field Lookup

```
Lemma lookup_field_in_value : ∀ v T i Ti,
  value v →
  has_type empty v T →
  Tlookup i T = Some Ti →
  ∃ vi, tlookup i v = Some vi ∧ has_type empty vi Ti.
```

+

Progress

Exercise: 3 stars (canonical_forms_of_arrow_types)

```
Lemma canonical_forms_of_arrow_types : ∀ Gamma s T1 T2,
     has_type Gamma s (TArrow T1 T2) →
     value s →
     ∃ x, ∃ S1, ∃ s2,
        s = tabs x S1 s2.
```

+

□

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

+

Theorem : For any term `t` and type `T`, if `empty |- t : T` then `t` is a value or `t ==> t'` for some term `t'`.

Proof: Let `t` and `T` be given such that `empty |- t : T`. We proceed by induction on the given typing derivation.

- The cases where the last step in the typing derivation is `T_Abs` or `T_RNil` are immediate because abstractions and `{}` are always values. The case for `T_Var` is vacuous because variables cannot be typed in the empty context.
- If the last step in the typing derivation is by `T_App`, then there are terms t_1 t_2 and types T_1 T_2 such that $t = t_1\ t_2$, $T = T_2$, `empty` $|- t_1 : T_1 \to T_2$ and `empty` $|- t_2 : T_1$.

  The induction hypotheses for these typing derivations yield that t_1 is a value or steps, and that t_2 is a value or steps.

  - Suppose t_1 ==> t_1' for some term t_1'. Then $t_1\ t_2$ ==> t_1' t_2 by `ST_App1`.
  - Otherwise t_1 is a value.
    - Suppose t_2 ==> t_2' for some term t_2'. Then $t_1\ t_2$ ==> $t_1\ t_2$' by rule `ST_App2` because t_1 is a value.
    - Otherwise, t_2 is a value. By Lemma `canonical_forms_for_arrow_types`, t_1 = \x:S_1.s2 for some x, S_1, and s_2. But then (\x:S_1.s2) t_2 ==> [x:=t_2]s_2 by `ST_AppAbs`, since t_2 is a value.
- If the last step of the derivation is by `T_Proj`, then there are a term `tr`, a type `Tr`, and a label `i` such that `t = tr.i`, `empty |- tr : Tr`, and `Tlookup i Tr = Some T`.

By the IH, either `tr` is a value or it steps. If `tr ==> tr'` for some term `tr'`, then `tr.i ==> tr'.i` by rule `ST_Proj1`.

If `tr` is a value, then Lemma `lookup_field_in_value` yields that there is a term `ti` such that `tlookup i tr = Some ti`. It follows that `tr.i ==> ti` by rule `ST_ProjRcd`.

- If the final step of the derivation is by `T_Sub`, then there is a type `S` such that `S <: T` and `empty |- t : S`. The desired result is exactly the induction hypothesis for the typing subderivation.
- If the final step of the derivation is by `T_RCons`, then there exist some terms t_1 `tr`, types T_1 `Tr` and a label `t` such that `t = {i=`t_1`, tr}`, `T = {i:`T_1`, Tr}`, `record_ty tr`, `record_tm Tr`, `empty |-` t_1 `:` T_1 and `empty |- tr : Tr`.

  The induction hypotheses for these typing derivations yield that t_1 is a value or steps, and that `tr` is a value or steps. We consider each case:

  - Suppose t_1 `==>` t_1`'` for some term t_1`'`. Then `{i=`t_1`, tr} ==> {i=`t_1`', tr}` by rule `ST_Rcd_Head`.
  - Otherwise t_1 is a value.
    - Suppose `tr ==> tr'` for some term `tr'`. Then `{i=`t_1`, tr} ==> {i=`t_1`, tr'}` by rule `ST_Rcd_Tail`, since t_1 is a value.
    - Otherwise, `tr` is also a value. So, `{i=`t_1`, tr}` is a value by `v_rcons`.

Inversion Lemmas

```
Lemma typing_inversion_var : ∀ Gamma x T,
  has_type Gamma (tvar x) T →
  ∃ S,
    Gamma x = Some S ∧ subtype S T.
```
+

```
Lemma typing_inversion_app : ∀ Gamma t1 t2 T2,
  has_type Gamma (tapp t1 t2) T2 →
  ∃ T1,
    has_type Gamma t1 (TArrow T1 T2) ∧
    has_type Gamma t2 T1.
```
+

```
Lemma typing_inversion_abs : ∀ Gamma x S1 t2 T,
     has_type Gamma (tabs x S1 t2) T →
     (∃ S2, subtype (TArrow S1 S2) T
              ∧ has_type (update Gamma x S1) t2 S2).
```

+

```
Lemma typing_inversion_proj : ∀ Gamma i t₁ Ti,
  has_type Gamma (tproj t₁ i) Ti →
  ∃ T, ∃ Si,
    Tlookup i T = Some Si ∧ subtype Si Ti ∧ has_type Gamma t₁ T.
```

+

```
Lemma typing_inversion_rcons : ∀ Gamma i ti tr T,
  has_type Gamma (trcons i ti tr) T →
  ∃ Si, ∃ Sr,
    subtype (TRCons i Si Sr) T ∧ has_type Gamma ti Si ∧
    record_tm tr ∧ has_type Gamma tr Sr.
```

+

```
Lemma abs_arrow : ∀ x S₁ s₂ T₁ T₂,
  has_type empty (tabs x S₁ s₂) (TArrow T₁ T₂) →
     subtype T₁ S₁
  ∧ has_type (update empty x S₁) s₂ T₂.
```

+

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 t tr,
      appears_free_in x t →
      appears_free_in x (trcons i t tr)
  | afi_rtail : ∀ x i t tr,
      appears_free_in x tr →
      appears_free_in x (trcons i t tr).

Hint Constructors appears_free_in.

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

+

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

+

Preservation

```
Lemma substitution_preserves_typing : ∀ Gamma x U v t S,
     has_type (update Gamma x U) t S →
     has_type empty v U →
     has_type Gamma ([x:=v]t) S.
```

+

```
Theorem preservation : ∀ t t' T,
     has_type empty t T →
     t ==> t' →
     has_type empty t' T.
```

+

Theorem: If `t`, `t'` are terms and `T` is a type such that `empty |- t : T` and `t ==> t'`, then `empty |- t' : T`.

Proof: Let `t` and `T` be given such that `empty |- t : T`. We go by induction on the structure of this typing derivation, leaving `t'` general. Cases `T_Abs` and `T_RNil` are vacuous because abstractions and `{}` don't step. Case `T_Var` is vacuous as well, since the context is empty.

- If the final step of the derivation is by `T_App`, then there are terms t_1 t_2 and types T_1 T_2 such that $t = t_1\ t_2$, $T = T_2$, empty $|- t_1 : T_1 \to T_2$ and empty $|- t_2 : T_1$.

  By inspection of the definition of the step relation, there are three ways $t_1\ t_2$ can step. Cases `ST_App1` and `ST_App2` follow immediately by the induction hypotheses for the typing subderivations and a use of `T_App`.

  Suppose instead $t_1\ t_2$ steps by `ST_AppAbs`. Then t_1 = `\x:S.t12` for some type `S` and term t_{12}, and $t' = [x:=t_2]t_{12}$.

  By Lemma `abs_arrow`, we have T_1 <: `S` and $x{:}S_1 |- s_2 : T_2$. It then follows by lemma `substitution_preserves_typing` that empty $|- [x:=t_2]\ t_{12} : T_2$ as desired.

- If the final step of the derivation is by `T_Proj`, then there is a term `tr`, type `Tr` and label `i` such that `t = tr.i`, `empty |- tr : Tr`, and `Tlookup i Tr = Some T`.

The IH for the typing derivation gives us that, for any term `tr'`, if `tr ==> tr'` then `empty |- tr' Tr`. Inspection of the definition of the step relation reveals that there are two ways a projection can step. Case `ST_Proj1` follows immediately by the IH.

Instead suppose `tr.i` steps by `ST_ProjRcd`. Then `tr` is a value and there is some term `vi` such that `tlookup i tr = Some vi` and `t' = vi`. But by lemma `lookup_field_in_value`, `empty |- vi : Ti` as desired.

- If the final step of the derivation is by `T_Sub`, then there is a type `S` such that `S <: T` and `empty |- t : S`. The result is immediate by the induction hypothesis for the typing subderivation and an application of `T_Sub`.
- If the final step of the derivation is by `T_RCons`, then there exist some terms t_1 `tr`, types T_1 `Tr` and a label `t` such that `t = {i=`t_1`, tr}`, `T = {i:`T_1`, Tr}`, `record_ty tr`, `record_tm Tr`, `empty |-` t_1 `:` T_1 and `empty |- tr : Tr`.

  By the definition of the step relation, `t` must have stepped by `ST_Rcd_Head` or `ST_Rcd_Tail`. In the first case, the result follows by the IH for t_1's typing derivation and `T_RCons`. In the second case, the result follows by the IH for `tr`'s typing derivation, `T_RCons`, and a use of the `step_preserves_record_tm` lemma.