# CIS 500 — Software Foundations
# Final Exam

## Answer key

December 18, 2007

## Operational semantics

The questions in this section concern a variant of the language of untyped arithmetic and boolean expressions that we used in the course. The original definition of this language is reproduced, for your reference, on page 16.

Suppose, now, that we define an alternate version of the single-step evaluation relation that drops the premise `nvalue n1` from the rules `E_PredSucc` and `E_IszeroSucc`:

```
Inductive alteval : tm -> tm -> Prop :=
  | AE_IfTrue : forall t1 t2,
        alteval (tm_if tm_true t1 t2)
                t1
  | AE_IfFalse : forall t1 t2,
        alteval (tm_if tm_false t1 t2)
                t2
  | AE_If : forall t1 t1' t2 t3,
        alteval t1 t1'
     -> alteval (tm_if t1 t2 t3)
                (tm_if t1' t2 t3)
  | AE_Succ : forall t1 t1',
        alteval t1 t1'
     -> alteval (tm_succ t1)
                (tm_succ t1')
  | AE_PredZero :
        alteval (tm_pred tm_zero)
                tm_zero
  | AE_PredSucc : forall t1,
        (* nvalue t1 -> *)
        alteval (tm_pred (tm_succ t1))
                t1
  | AE_Pred : forall t1 t1',
        alteval t1 t1'
     -> alteval (tm_pred t1)
                (tm_pred t1')
  | AE_IszeroZero :
        alteval (tm_iszero tm_zero)
                tm_true
  | AE_IszeroSucc : forall t1,
        (* nvalue t1 -> *)
        alteval (tm_iszero (tm_succ t1))
                tm_false
  | AE_Iszero : forall t1 t1',
        alteval t1 t1'
     -> alteval (tm_iszero t1)
                (tm_iszero t1').
```

1. (2 points) Is the alternate evaluation relation deterministic? That is, is the following lemma provable? (Write "yes" or "no." If you write "no," give a counter-example.)

```
    Lemma alteval_deterministic : forall t t' t'',
         alteval t t'
      -> alteval t t''
      -> t' = t''.
```

   *Answer: No. A counter-example is* `t = pred (succ (pred zero))`

2. (3 points) Can every *value* that results from applying `eval` many times also be obtained by applying `alteval` many times?

   Formally, if we define

```
    Notation evalmany    := (refl_trans_closure _ eval).
    Notation altevalmany := (refl_trans_closure _ alteval).
```

   then is the lemma

```
    Lemma eval_result__alteval_result : forall t t',
         evalmany t t'
      -> value t'
      -> altevalmany t t'.
```

   provable? Write "yes" or "no." If you write "no," give a counter-example.

   *Answer: Yes.*

3. (3 points) Conversely, can every value that results from applying `alteval` many times also be obtained by applying `eval` many times? Write "yes" or "no." If you write "no," give a counter-example.

```
    Lemma alteval_result__eval_result : forall t t',
         altevalmany t t'
      -> value t'
      -> evalmany t t'.
```

   *Answer: No. A counter-example is* `t = tm_iszero (tm_succ tm_true)`.

4. (12 points) Complete the following `Fixpoint` definition so that it defines a functional version of the `alteval` relation — i.e., so that `alt_simplify_step t = Some _ t'` whenever `alteval t t'` holds and `alt_simplify_step t = None _` if there is no `t'` such that `alteval t t'`.

```
Fixpoint alt_simplify_step (t:tm) struct t : option tm :=
  match t with
  | tm_if t1 t2 t3 =>
      match alt_simplify_step t1 with
      | None => match t1 with
                | tm_true => Some _ t2
                | tm_false => Some _ t3
                | _ => None _
                end
      | Some t1' => Some _ (tm_if t1' t2 t3)
      end
  | tm_succ t1 =>
      match alt_simplify_step t1 with
      | None => None _
      | Some t1' => Some _ (tm_succ t1')
      end
  | tm_pred t1 =>
      (* FILL IN HERE: *)
      match alt_simplify_step t1 with
      | None => match t1 with
                | tm_zero => Some _ tm_zero
                | tm_succ t2 => Some _ t2
                end
      | Some t1' => Some _ (tm_pred t1')
      end
  | tm_iszero t1 =>
      (* FILL IN HERE: *)
      match alt_simplify_step t1 with
      | None => match t1 with
                | tm_zero => Some _ tm_true
                | tm_succ t2 => Some _ tm_false
                end
      | Some t1' => Some _ (tm_iszero t1')
      end
  | _ =>
      (* FILL IN HERE: *)
      None _
  end.
```

*Grading scheme:*

- *7 points for the first blank, 3 for the second, 2 for the last*
- *-1 for minor mistakes*
- *-5 for interchanging `None` and `Some` cases*

## Untyped Lambda-Calculus

The following questions are about the untyped lambda calculus. For reference, the definition of this language and names for several specific lambda-terms (`c_zero`, `pls`, etc., etc.) appear on page 18.

5. (4 points) Here are the definitions of `pls`, `c_one`, and `c_two`:

```
Notation pls   := (\m, \n, \s, \z, m @ s @ (n @ s @ z)).
Notation c_one := (\s, \z, s @ z).
Notation c_two := (\s, \z, s @ (s @ z)).
```

Write the normal form of the term (`pls @ c_one @ c_two`).

*Answer:*
*`\s, \z, c_one @ s @ (c_two @ s @ z),`*
*since*
*`(\m, \n, \s, \z, m @ s @ (n @ s @ z)) @ c_one @ c_two` $\longrightarrow$*
*`(\n, \s, \z, c_one @ s @ (n @ s @ z)) @ c_two` $\longrightarrow$*
*`\s, \z, c_one @ s @ (c_two @ s @ z)`*

Recall the definitions of observational and behavioral equivalence in the untyped lambda-calculus:

- Two terms `s` and `t` are *observationally equivalent* iff either both are normalizable (i.e., they reach a normal form after a finite number of evaluation steps) or both are divergent.
- Terms `s` and `t` are *behaviorally equivalent* iff, for every finite list of closed values `[v_1, v_2, ..., v_n]` (including the empty list), the applications

  ```
  s @ v_1 @ v_2 ... @ v_n
  ```

  and

  ```
  t @ v_1 @ v_2 ... @ v_n
  ```

  are observationally equivalent.

6. (9 points) Let us write "`s` and `t` are distinguished by `[v_1, v_2, ..., v_n]`" to mean that

```
s @ v_1 @ v_2 ... @ v_n
```

and

```
t @ v_1 @ v_2 ... @ v_n
```

are *not* observationally equivalent.

(a) Consider the following pair of terms:

```
s = \x, \y, x
t = \x, \y, y
```

Circle one of the following choices:

i. `s` and `t` are behaviorally equivalent
ii. `s` and `t` are distinguished by `[poisonpill]`
iii. `s` and `t` are distinguished by `[tru, fls]`
iv. `s` and `t` are distinguished by `[poisonpill, tru]`
v. `s` and `t` are distinguished by `[tru, poisonpill]`
vi. `s` and `t` are distinguished by `[poisonpill, tru, fls]`
vii. `s` and `t` are distinguished by `[tru, poisonpill, fls]`
viii. `s` and `t` are distinguished by some *other* list of closed values

*Answer: vi*

(b) Consider the following pair of terms:

```
s = (\x, x @ x) @ (\x, x)
t = \x, \y, y
```

Circle one of the following choices:

i. `s` and `t` are behaviorally equivalent
ii. `s` and `t` are distinguished by `[]` (the empty list)
iii. `s` and `t` are distinguished by `[poisonpill]`
iv. `s` and `t` are distinguished by `[tru, fls]`
v. `s` and `t` are distinguished by `[poisonpill, tru]`
vi. `s` and `t` are distinguished by `[tru, poisonpill]`
vii. `s` and `t` are distinguished by some *other* list of closed values

*Answer: v*

(c) Consider the following pair of terms:

```
s = \s, \z, s @ (s @ z)
t = \s, \z, s @ (s @ (s @ z))
```

Circle one of the following choices:

i. `s` and `t` are behaviorally equivalent
ii. `s` and `t` are distinguished by `[]` (the empty list)
iii. `s` and `t` are distinguished by `[poisonpill]`
iv. `s` and `t` are distinguished by `[poisonpill, tru]`
v. `s` and `t` are distinguished by `[tru, poisonpill]`
vi. `s` and `t` are distinguished by `[(\x,poisonpill)]`
vii. `s` and `t` are distinguished by `[(\x,\x,\x,poisonpill)]`
viii. `s` and `t` are distinguished by some *other* list of closed values

*Answer: viii*

## Subtyping

The questions in this section concern the simply typed lambda-calculus with records and subtyping. For reference, the definition of this language appears on page 20.

7. (7 points) Suppose we have types `S`, `T`, `U`, and `V` with `S <: T` and `U <: V`. Which of the following subtyping assertions are then true? (Circle T or F for each.)

   (a) `T-->S <: T-->S`

   [✓ T] F

   (b) `Top-->U <: S-->Top`

   [✓ T] F

   (c) `T-->T-->U <: S-->S-->V`

   [✓ T] F

   (d) `(T-->T)-->U <: (S-->S)-->V`

   T [✓ F]

   (e) `((T-->S)-->T)-->U <: ((S-->T)-->S)-->V`

   [✓ T] F

   (f) `[[a~S, b~V]] <: [[a~T, B~U]]`

   T [✓ F]

   (g) `[[a~S; S]] <: [[a~T; Top]]`

   T [✓ F]

8. (4 points) What is the *smallest* type `T` that makes the following assertion true?

```
      empty |- (\x~Top, x) @ [|a==(\z~A,z),b==(\z~B,z)|] ~ T
```

*Answer:*

```
T  =  Top
```

9. (4 points) What is the *smallest* type `T` that makes the following assertion true?

```
    exists S,
      empty |- (\r~[[x~A; T]], (r#y) @ (r#x)) ~ S
```

*Answer:* There is none – *any* record type that has at least a field `y` (of appropriate type) will make the assertion true.

10. (4 points) What is the *largest* type `T` that makes the same assertion true?

```
    exists S,
      empty |- (\r~[[x~A; T]], (r#y) @ (r#x)) ~ S
```

*Answer:*

```
T  =  [[y~A-->Top]]
```

*Grading scheme: 2 points for some valid type*

11. (7 points) How many supertypes does the type

```
   [[ x~A, y~C->C ]]
```

have? That is, how many different types `T` are there such that `[[ x~A, y~C->C ]] <: T`?

(We consider two types to be *different* if they are written differently, even if each is a subtype of the other. For example, `[[x~A,y~B]]` and `[[y~B,x~A]]` are different.)

*Answer:* I count 19...

```
   [[ x~A, y~C->C ]]                [[ y~Top, x~A ]]
   [[ x~Top, y~C->C ]]              [[ y~Top, x~Top ]]
   [[ x~A, y~C->Top ]]              [[ x~A ]]
   [[ x~Top, y~C->Top ]]            [[ x~Top ]]
   [[ x~A, y~Top ]]                 [[ y~C->C ]]
   [[ x~Top, y~Top ]]               [[ y~C->Top ]]
   [[ y~C->C, x~A ]]                [[ y~Top ]]
   [[ y~C->C, x~Top ]]              [[ ]]
   [[ y~C->Top, x~A ]]              Top
   [[ y~C->Top, x~Top ]]
```

- 19 *rightarrow* 7 points
- 12-18 *rightarrow* 5 points (only if examples are shown)
- 5-11 *rightarrow* 3 points
- $\leq$4 or infinite *rightarrow* 0 points

12. (10 points) Recall the following properties of the simply typed lambda-calculus with subtyping:

```
Theorem preservation : forall t t' T,
     empty |- t ~ T
  -> eval t t'
  -> empty |- t' ~ T.

Theorem progress : forall t T,
     empty |- t ~ T
  -> value t \/ exists t', eval t t'.
```

Each part of this problem suggests a different way of changing the language. (These changes are not cumulative: each part starts from the original language.) In each part, indicate (by circling TRUE or FALSE) whether each property remains true or becomes false after the suggested change. If a property becomes false, give a counterexample.

(a) Suppose we add the following evaluation rule:

```
| E_Funny : forall t,
       eval ([||] @ t) (t @ [||]).
```

Progress: *Answer: True*

Preservation: *Answer: True*

(b) Suppose we add the same evaluation rule

```
| E_Funny : forall t,
       eval ([||] @ t) (t @ [||]).
```

*and* the following new typing rule:

```
| T_Funny : forall t,
       empty |- [||] ~ Top-->Top
```

Progress: *Answer: True*

Preservation: *Answer: False: For example,* `[||] @ (\x~A,x)` *has type* `Top`*, but it steps to* `(\x~A,x) @ [||]`*, which is ill-typed.*

*Grading scheme: 2 points for correct True/False answers; 2 points for correct counterexample.*

13. (5 points) Is the following statement true or false? Briefly explain your answer.

```
forall T,
     ~(exists n, T = ty_base n)
  -> exists S,
        S <: T  /\  S <> T.
```

*Answer: False:* `T = Top-->A` *is a counterexample.*

14. (16 points) Below we give the complete Coq proof for the key technical lemma `sub_inversion_arrow`. Add an informal explanation of the proof, using the blanks provided. Your explanation should give a sufficiently clear understanding of the proof that, if we took just the explanation and gave it to a skilled Coq user, they could easily reproduce the formal proof. You should *not* include low-level details that such a user could easily fill in for themselves. (Obviously, this is a matter of taste to some extent. We will take this into account when grading.)

```
Lemma sub_inversion_arrow : forall U V1 V2,
     U <: V1 --> V2
  -> exists U1, exists U2, (U=U1-->U2) /\ (V1<:U1) /\ (U2<:V2).
Proof.
  (* By induction on a derivation of U <: V1 --> V2.  Since the
     right-hand side is an arrow type, most of the subtyping rules
     cannot apply; there are just three cases to consider. *)
  intros U V1 V2 Hs.
  remember (V1-->V2) as V.
  generalize dependent V2. generalize dependent V1.
  (subtyping_cases (induction Hs) CASE);
         subst; intros; try solve [solve by inversion].
    CASE "S_Refl".
      (* If the final rule is S_Refl, then U = V1-->V2 and
         we can choose U1=V1 and U2=V2; the result then follows by
         S_Refl. *)
      apply ex_intro with (witness:=V1). apply ex_intro with (witness:=V2).
      subst. inversion H.
      apply conj. reflexivity. apply conj; apply S_Refl; assumption.
    CASE "S_Trans".
      (* If the final rule is S_Trans, then there is some S such
         that U <: S and S <: V1-->V2.  By the IH, there are some S1
         and S2 such that S=S1-->S2, V1<:S1, and S2<:V2.  By the IH
         again, there are some U1 and U2 such that U=U1-->U2, S1<:U1,
         and U2<:S2.  By S_Trans, V1<:U1 and U2<:V2. *)
      apply IHHs2 in HeqV. destruct HeqV. destruct H.
      rename witness into U1. rename witness0 into U2.
      destruct H. destruct H0.
      apply IHHs1 in H. destruct H. destruct H.
      rename witness into S1. rename witness0 into S2.
      destruct H. destruct H2.
      apply ex_intro with (witness := S1).
      apply ex_intro with (witness := S2).
      apply conj. assumption. apply conj.
      apply S_Trans with (U := U1); assumption.
      apply S_Trans with (U := U2); assumption.
    CASE "S_Arrow".
      (* If the final rule is S_Arrow, then U=U1-->U2, with V1<:U1
         and U2<:V2, which is exactly what we need. *)
      apply ex_intro with (witness := S1).
      apply ex_intro with (witness := S2).
      inversion HeqV. subst.
      auto using conj.
Qed.
```

*Grading scheme:*

- *3 points for the first blank.*
  - *+3 for induction on the right thing*
  - *-1/2 for miscellaneous wrong stuff*
- *3 points for the second blank.*
  - *-1 for no* `V1-->V2`
  - *-1 for no* `S_Refl`
  - *-1/2 for miscellaneous wrong stuff*
- *7 points for the third blank.*
  - *-1 – about S*
  - *-1 – IH details (each)*
  - *-1 for trans*
  - *-5 for major confusion*
  - *-1/2 for miscellaneous wrong stuff*
- *3 points for the fourth blank.*
  - *-1 for wrong set up*
  - *-1/2 for miscellaneous wrong stuff*

15. (7 points) Among the technical lemmas appearing in the theory of the simply typed lambda-calculus with subtyping is this one:

```
Lemma subtypes__well_formed : forall S T,
     S <: T
  -> well_formed S /\ well_formed T.
```

The proof of this lemma goes by induction on the derivation of `S <: T`.

Now suppose we change the statement of the lemma to

```
Lemma subtypes__well_formed : forall S T,
     S <: T
  -> well_formed T.
```

and again start the proof by induction on the derivation of `S <: T`. Will we succeed in completing the proof? Briefly explain.

*Answer: No: The proof will get stuck in the `S_Arrow` case, where `T = T1-->T2` and `S = S1-->S2`. The induction principle gives us the induction hypotheses `well_formed S1` and `well_formed T2`, from which we* cannot *assemble a proof of `well_formed T1-->T2`. Grading scheme:*

- *7 points for "No" plus mentioning `S_Arrow`*
- *4 points for "No" with a vague but not incorrect explanation*
- *3 points for "No" with a bogus explanation*
- *1 point for "Yes"*

## Algorithmic Subtyping

16. (7 points) The "declarative" typing relation `Gamma |- t ~ T` defined on page 20 cannot be translated directly ("clause for clause") into a recursive `Fixpoint` definition that, given `Gamma` and `t`, returns a `T` such that `Gamma |- t ~ T` (or returns `None` to indicate that no such `T` exists). Explain why not.

    *Answer: The `T_Sub` rule overlaps with all of the other rules, so a "direct" clause-for-clause implementation would have to involve backtracking. Worse, the function clause corresponding to the `T_Sub` would make a recursive call with the same arguments, causing an infinite loop. Grading scheme:*

    - *7 points for either correct reason*
    - *4 points for mentioning that `T_App` is a problem / relation not a function / variable not bound*

## Featherweight Java

The definition of Featherweight Java is summarized on page 24 at the end.

17. (8 points) The preservation theorem for Featherweight Java is *not* stated like this:

```
Theorem preservation : forall CT Gamma t t' C,
     CT >> Gamma |- t ~ C
  -> eval CT t t'
  -> CT >> Gamma |- t' ~ C.
```

(1) Explain what is wrong, (2) give an example FJ term that breaks this version of preservation, and (3) state the theorem correctly.

For (2), you may assume that the class table defines the classes `A`, `B`, and `Pair` (all extending `Object`), as in the lecture and homework assignment.

*Answer: This version of the theorem is false because the types of programs can strictly decrease (in the subtype relation) during evaluation. For instance, if the class table includes a class `A`, then*

```
(Object)(new A())
```

*has type `Object` but steps to `new A()`, which does not have type `Object` (it only has type `A`). Here is the good version:*

```
Theorem preservation : forall CT Gamma t t' C,
     CT >> Gamma |- t ~ C
  -> eval CT t t'
  -> exists C', (subtyping CT C' C) /\ (CT >> Gamma |- t' ~ C').
```

*Grading scheme:*

- *2 points for part 1*
- *3 points for part 2*
- *3 points for part 3 (one for mentioning `exists`, one for mentioning subtyping, and one more for being completely correct)*

## Coq

Recall that, in Coq, the notation "`exists x, P`" is actually a shorthand for "`ex _ (fun x=>P)`", where the proposition `ex` is defined like this:

```
Inductive ex (X : Type) (P : X -> Prop) : Prop :=
  ex_intro : forall witness:X, P witness -> ex X P.
```

18. (4 points) Similarly, the notation "`A /\ B`" is a shorthand for "`and A B`", where `and` is an inductively defined proposition. Fill in the blank to complete the definition of `and`.

    *Answer:*

    ```
    Inductive and (A B : Prop) : Prop :=
      conj : A -> B -> (and A B).
    ```

    *Grading scheme:*

    - *-1 for no constructor*
    - *-1 for introducing `forall`*
    - *-2 for introducing vars*

19. (4 points) The notation "`A \/ B`" is a shorthand for "`or A B`", where `or` is an inductively defined proposition. Fill in the blank to complete its definition.

    *Answer:*

    ```
    Inductive or (A B : Prop) : Prop :=
      | or_introl : A -> or A B
      | or_intror : B -> or A B.
    ```

    *Grading scheme:*

    - *-4 for including only one case*
    - *-1 if no constructor*
    - *-2 for introducing vars*
    - *-3 for `True` instead of `or A B`, given everything else is correct*

## For reference: Boolean and arithmetic expressions

```
Inductive tm : Set :=
  | tm_true : tm
  | tm_false : tm
  | tm_if : tm -> tm -> tm -> tm
  | tm_zero : tm
  | tm_succ : tm -> tm
  | tm_pred : tm -> tm
  | tm_iszero : tm -> tm.

Inductive bvalue : tm -> Prop :=
  | bv_true : bvalue tm_true
  | bv_false : bvalue tm_false.

Inductive nvalue : tm -> Prop :=
  | nv_zero : nvalue tm_zero
  | nv_succ : forall t, nvalue t -> nvalue (tm_succ t).

Definition value (t:tm) := bvalue t \/ nvalue t.

Inductive eval : tm -> tm -> Prop :=
  | E_IfTrue : forall t1 t2,
        eval (tm_if tm_true t1 t2)
             t1
  | E_IfFalse : forall t1 t2,
        eval (tm_if tm_false t1 t2)
             t2
  | E_If : forall t1 t1' t2 t3,
        eval t1 t1'
     -> eval (tm_if t1 t2 t3)
             (tm_if t1' t2 t3)
  | E_Succ : forall t1 t1',
        eval t1 t1'
     -> eval (tm_succ t1)
             (tm_succ t1')
  | E_PredZero :
        eval (tm_pred tm_zero)
             tm_zero
  | E_PredSucc : forall t1,
        nvalue t1
     -> eval (tm_pred (tm_succ t1))
             t1
  | E_Pred : forall t1 t1',
        eval t1 t1'
     -> eval (tm_pred t1)
             (tm_pred t1')
  | E_IszeroZero :
        eval (tm_iszero tm_zero)
             tm_true
  | E_IszeroSucc : forall t1,
        nvalue t1
     -> eval (tm_iszero (tm_succ t1))
             tm_false
  | E_Iszero : forall t1 t1',
        eval t1 t1'
     -> eval (tm_iszero t1)
             (tm_iszero t1').

Inductive ty : Set :=
  | ty_bool : ty
  | ty_nat : ty.

Inductive has_type : tm -> ty -> Prop :=
  | T_True :
         has_type tm_true ty_bool
  | T_False :
         has_type tm_false ty_bool
  | T_If : forall t1 t2 t3 T,
         has_type t1 ty_bool
      -> has_type t2 T
      -> has_type t3 T
      -> has_type (tm_if t1 t2 t3) T
  | T_Zero :
         has_type tm_zero ty_nat
  | T_Succ : forall t1,
         has_type t1 ty_nat
      -> has_type (tm_succ t1) ty_nat
  | T_Pred : forall t1,
         has_type t1 ty_nat
      -> has_type (tm_pred t1) ty_nat
  | T_Iszero : forall t1,
         has_type t1 ty_nat
      -> has_type (tm_iszero t1) ty_bool.
```

## For reference: Untyped lambda-calculus

```
Definition name := nat.

Inductive tm : Set :=
  | tm_const : name -> tm
  | tm_var : name -> tm
  | tm_app : tm -> tm -> tm
  | tm_abs : name -> tm -> tm.

Notation "‘ n" := (tm_const n) (at level 19).
Notation "! n" := (tm_var n) (at level 19).
Notation "\ x , t" := (tm_abs x t) (at level 21).
Notation "r @ s" := (tm_app r s) (at level 20).

Fixpoint only_constants (t:tm) {struct t} : yesno :=
  match t with
  | tm_const _ => yes
  | tm_app t1 t2 => both_yes (only_constants t1) (only_constants t2)
  | _ => no
  end.

Inductive value : tm -> Prop :=
  | v_const : forall t,
      only_constants t = yes -> value t
  | v_abs : forall x t,
      value (\x, t).

Fixpoint subst (x:name) (s:tm) (t:tm) {struct t} : tm :=
  match t with
  | ‘c => ‘c
  | !y => if eqname x y then s else t
  | \y, t1 => if eqname x y then t else (\y, subst x s t1)
  | t1 @ t2 => (subst x s t1) @ (subst x s t2)
  end.

Inductive eval : tm -> tm -> Prop :=
  | E_AppAbs : forall x t12 v2,
         value v2
      -> eval ((\x, t12) @ v2) ({x |-> v2} t12)
  | E_App1 : forall t1 t1’ t2,
         eval t1 t1’
      -> eval (t1 @ t2) (t1’ @ t2)
  | E_App2 : forall v1 t2 t2’,
         value v1
      -> eval t2 t2’
      -> eval (v1 @ t2) (v1 @ t2’).
```

```
Notation tru := (\t, \f, t).
Notation fls := (\t, \f, f).

Notation bnot := (\b, b @ fls @ tru).
Notation and := (\b, \c, b @ c @ fls).
Notation or := (\b, \c, b @ tru @ c).
Notation test :=  (\b, \t, \f, b @ t @ f @ (\x,x)).

Notation pair := (\f, \s, (\b, b @ f @ s)).
Notation fst  := (\p, p @ tru).
Notation snd  := (\p, p @ fls).

Notation c_zero  := (\s, \z, z).
Notation c_one   := (\s, \z, s @ z).
Notation c_two   := (\s, \z, s @ (s @ z)).
Notation c_three := (\s, \z, s @ (s @ (s @ z))).
Notation scc := (\n, \s, \z, s @ (n @ s @ z)).
Notation pls := (\m, \n, \s, \z, m @ s @ (n @ s @ z)).
Notation tms := (\m, \n, m @ (pls @ n) @ c_zero).
Notation iszro := (\m, m @ (\x, fls) @ tru).
Notation zz := (pair @ c_zero @ c_zero).
Notation ss := (\p, pair @ (snd @ p) @ (pls @ c_one @ (snd @ p))).
Notation prd := (\m, fst @ (m @ ss @ zz)).

Notation omega := ((\x, x @ x) @ (\x, x @ x)).
Notation poisonpill := (\y, omega).

Notation Z := (\f,
                 (\y,   (\x, f @ (\y, x @ x @ y))
                      @ (\x, f @ (\y, x @ x @ y))
                      @ y)).
Notation f_fact := (\f,
                        \n,
                           test
                             @ (iszro @ n)
                             @ (\z, c_one)
                             @ (\z, tms @ n @ (f @ (prd @ n)))).

Notation fact := (Z @ f_fact).
```

**For reference: Simply Typed Lambda-Calculus with Records and Subtyping**

```
Inductive ty : Set :=
  | ty_top      : ty
  | ty_base     : nat -> ty
  | ty_arrow    : ty -> ty -> ty
  | ty_rcd_nil  : ty
  | ty_rcd_cons : nat -> ty -> ty -> ty.

Inductive tm : Set :=
  | tm_var : nat -> tm
  | tm_app : tm -> tm -> tm
  | tm_abs : nat -> ty -> tm -> tm
  | tm_rcd_nil : tm
  | tm_rcd_cons : nat -> tm -> tm -> tm
  | tm_proj : tm -> nat -> tm.

Notation A := (ty_base one).
Notation B := (ty_base two).
Notation C := (ty_base three).
Notation Top := ty_top.
Notation "S --> T" := (ty_arrow S T) (at level 20, right associativity).
Notation "[[ ]]" := (ty_rcd_nil).
Notation "[[ l1 ~ T1 ; T2 ]]" := (ty_rcd_cons l1 T1 T2).

Notation "! n" := (tm_var n) (at level 39).
Notation "\ x ~ T , t" := (tm_abs x T t) (at level 42).
Notation "r @ s" := (tm_app r s) (at level 40, left associativity).
Notation "r # s" := (tm_proj r s) (at level 41).
Notation "[| |]" := (tm_rcd_nil).
Notation "[| l1 == t1 ; t2 |]" := (tm_rcd_cons l1 t1 t2).

Notation "[[ l1 ~ T1 ]]" := ( [[ l1~T1; [[]] ]]).
Notation "[[ l1 ~ T1 , l2 ~ T2 ]]" := ( [[ l1~T1; [[l2~T2]] ]]).
Notation "[[ l1 ~ T1 , l2 ~ T2 , l3 ~ T3 ]]" := ( [[ l1~T1; [[l2~T2,l3~T3]] ]]).
Notation "[| l1 == T1 |]" := ([| l1==T1; [||] |]).
Notation "[| l1 == T1 , l2 == T2 |]" := ([| l1==T1; [|l2==T2|] |]).
Notation "[| l1 == T1 , l2 == T2 , l3 == T3 |]" := ([| l1==T1; [|l2==T2,l3==T3|] |]).

Fixpoint subst (x:nat) (s:tm) (t:tm) {struct t} : tm :=
  match t with
  | !y => if eqnat x y then s else t
  | \y~T, t1 => if eqnat x y then t else \y~T, {x |-> s}t1
  | t1 @ t2 => ({x |-> s}t1) @ ({x |-> s}t2)
  | [||] => [||]
  | [| l==t1; t2 |] => [| l=={x |-> s}t1; {x |-> s}t2 |]
  | t # k => ({x |-> s}t) # k
  end

where "{ x |-> s } t" := (subst x s t).
```

```
Inductive value : tm -> Prop :=
  | v_abs : forall x T t,
        value (\x~T, t)
  | v_rcd_nil :
        value [||]
  | v_rcd_cons : forall l t1 t2,
        value t1
      -> value t2
      -> value [| l==t1; t2|].

Inductive eval : tm -> tm -> Prop :=
  | E_AppAbs : forall x T t12 v2,
        value v2
      -> eval ((\x~T, t12) @ v2) ({x |-> v2} t12)
  | E_App1 : forall t1 t1' t2,
        eval t1 t1'
      -> eval (t1 @ t2) (t1' @ t2)
  | E_App2 : forall v1 t2 t2',
        value v1
      -> eval t2 t2'
      -> eval (v1 @ t2) (v1 @ t2')
  | E_Rcdcons1 : forall k t1 t1' t2,
        eval t1 t1'
      -> eval [|k==t1;t2|] [|k==t1';t2|]
  | E_Rcdcons2 : forall k t1 t2 t2',
        value t1
      -> eval t2 t2'
      -> eval [|k==t1;t2|] [|k==t1;t2'|]
  | E_ProjRcdcons1 : forall k t1 t2,
        value t1
      -> value t2
      -> eval ([|k==t1;t2|] # k) t1
  | E_ProjRcdcons2 : forall k k' t1 t2,
        value t1
      -> value t2
      -> k <> k'
      -> eval ([|k'==t1;t2|] # k) (t2 # k)
  | E_Proj : forall k t t',
        eval t t'
      -> eval (t # k) (t' # k).

Inductive doesn't_bind (k:nat) : ty -> Prop :=
  | db_nil :
        doesn't_bind k [[]]
  | db_cons : forall k' T1 T2,
        k <> k'
      -> doesn't_bind k T2
      -> doesn't_bind k [[k'~T1;T2]].

Inductive record_type : ty -> Prop :=
  | rt_nil :
        record_type [[]]
  | rt_cons : forall k T1 T2,
         record_type [[k~T1;T2]].

Inductive well_formed : ty -> Prop :=
  | wf_top :
         well_formed Top
  | wf_base : forall n,
         well_formed (ty_base n)
  | wf_arrow : forall T1 T2,
         well_formed T1
      -> well_formed T2
      -> well_formed (T1-->T2)
  | wf_rcdnil :
         well_formed [[]]
  | wf_rcdcons : forall k T1 T2,
         well_formed T1
      -> well_formed T2
      -> record_type T2
      -> doesn't_bind k T2
      -> well_formed [[k~T1;T2]].

Inductive subtyping : ty -> ty -> Prop :=
  | S_Refl : forall T,
         well_formed T
      -> T <: T
  | S_Trans : forall S U T,
         S <: U
      -> U <: T
      -> S <: T
  | S_Top : forall S,
         well_formed S
      -> S <: Top
  | S_Arrow : forall S1 S2 T1 T2,
         T1 <: S1
      -> S2 <: T2
      -> S1-->S2 <: T1-->T2
  | S_Rcdwidth : forall k T1 T2,
         well_formed [[k~T1;T2]]
      -> [[k~T1;T2]] <: [[]]
  | S_Rcddepth : forall k S1 S2 T1 T2,
         S1 <: T1
      -> S2 <: T2
      -> well_formed [[k~S1;S2]]
      -> well_formed [[k~T1;T2]]
      -> [[k~S1;S2]] <: [[k~T1;T2]]
  | S_Rcdperm : forall k1 k2 S1 S2 S3,
         well_formed [[k1~S1; [[k2~S2; S3]] ]]
      -> k1 <> k2
      -> [[k1~S1; [[k2~S2; S3]] ]] <: [[k2~S2; [[k1~S1; S3]] ]]

where "S <: T" := (subtyping S T).
```

```
Notation context := (alist ty).

Definition empty : context := nil _.

Fixpoint ty_rcd_lookup (k:nat) (t:ty) {struct t} : option ty :=
  match t with
  | ty_rcd_cons k' T' t' =>
      if eqnat k k' then Some _ T' else ty_rcd_lookup k t'
  | _ =>
      None _
  end.

Definition ty_rcd_binds (k:nat) (Tk:ty) (T:ty) :=
  ty_rcd_lookup k T = Some _ Tk.

Inductive typing : context -> tm -> ty -> Prop :=
  | T_Var : forall Gamma x T,
         binds _ x T Gamma
      -> well_formed T
      -> Gamma |- (!x) ~ T
  | T_Abs : forall Gamma x T1 T2 t,
         well_formed T1
      -> [(x,T1)] ++ Gamma |- t ~ T2
      -> Gamma |- (\x~T1, t) ~ T1-->T2
  | T_App : forall S T Gamma t1 t2,
         Gamma |- t1 ~ (S-->T)
      -> Gamma |- t2 ~ S
      -> Gamma |- (t1 @ t2) ~ T
  | T_Rcdnil : forall Gamma,
         Gamma |- [||] ~ [[]]
  | T_Rcdcons : forall Gamma k t1 t2 T1 T2,
         Gamma |- t1 ~ T1
      -> Gamma |- t2 ~ T2
      -> well_formed [[k~T1;T2]]
      -> Gamma |- [|k==t1;t2|] ~ [[k~T1;T2]]
  | T_Proj : forall Gamma k Tk t T,
         Gamma |- t ~ T
      -> ty_rcd_binds k Tk T
      -> Gamma |- t # k ~ Tk
  | T_Sub : forall Gamma t S T,
         Gamma |- t ~ S
      -> S <: T
      -> Gamma |- t ~ T

where "Gamma |- t ~ T" := (typing Gamma t T).
```

## For reference: Featherweight Java

```
Definition varName    :Set := nat.
Definition fieldName  :Set := nat.
Definition methodName :Set := nat.
Definition className  :Set := nat.

Inductive tm : Set :=
  | tm_var : varName -> tm                        (* variable *)
  | tm_field : tm -> fieldName -> tm              (* field access *)
  | tm_invoke : tm -> methodName -> list tm -> tm (* method invocation *)
  | tm_new : className -> list tm -> tm           (* object creation *)
  | tm_cast : className -> tm -> tm.              (* cast *)

Definition this := zero.

Inductive value : tm -> Prop :=
  | v_new : forall C vl,
          value_list vl
      -> value (tm_new C vl)
with value_list : list tm -> Prop :=
  | v_nil : value_list (nil _)
  | v_cons : forall v l,
          value v
      -> value_list l
      -> value_list (v :: l).

Inductive K : Set :=
  | constructor :
         className -> list (varName * className)
      -> list varName -> list varName -> K.

Inductive M : Set :=
  | method :
         className -> methodName
      -> list (varName * className) -> tm -> M.

Inductive CL : Set :=
  | class :
         className -> className
      -> list (fieldName * className) -> K -> list M -> CL.

Definition CT : Set := alist CL.

Definition Object :=  zero.

Inductive subtyping : CT -> className -> className -> Prop :=
  | S_Refl : forall CT C,
         subtyping CT C C
  | S_Trans : forall CT C D E,
         subtyping CT C D
      -> subtyping CT D E
      -> subtyping CT C E
  | S_Ext : forall CT C D Cf K M,
         lookup _ C CT = Some _ (class C D Cf K M)
      -> subtyping CT C D.

(* Field lookup *)
Inductive fields : CT -> className -> list (fieldName * className) -> Prop :=
   | f_obj : forall CT C,
          C = Object -> fields CT C (nil _)
   | f_class : forall CT C D Cf K M' Dg,
          lookup _ C CT = Some _ (class C D Cf K M')
     -> fields CT D Dg
     -> fields CT C  (Dg ++ Cf).

(* Search for a method in a list of method delarations *)
Fixpoint mlookup (m : methodName) (l : list M) {struct l} : option M :=
  match l with
  | nil => None _
  | (method T m' Cx t) :: l' =>
      if eqnat m m' then Some _ (method T m' Cx t) else mlookup m l'
  end.

(* Method type lookup *)
Inductive mtype : CT -> methodName -> className -> list className -> className -> Prop :=
   | mt_class : forall CT m C D Cf K M' Cx t B' B,
         lookup _ C CT = Some _ (class C D Cf K M')
     -> mlookup m M' = Some _ (method B m Cx t)
     -> B' = map _ _ (fun p => match p with (f,s) => s end) Cx
     -> mtype CT m C B' B
   | mt_super : forall CT m C D Cf K M' B' B,
         lookup _ C CT = Some _ (class C D Cf K M')
     -> mlookup m M' = None _
     -> mtype CT m D B' B
     -> mtype CT m C B' B.

(* Method body lookup *)
Inductive mbody : CT -> methodName -> className -> list varName -> tm -> Prop :=
   | mb_class : forall CT m C D Cf K M' Cx t x B,
         lookup _ C CT = Some _ (class C D Cf K M')
     -> mlookup m M' = Some _ (method B m Cx t)
     -> x = map _ _ (fun p => match p with (f,s) => f end) Cx
     -> mbody CT m C x t
   | mb_super : forall CT m C D Cf K M' x t,
         lookup _ C CT = Some _ (class C D Cf K M')
     -> mlookup m M' = None _
     -> mbody CT m D x t
     -> mbody CT m C x t.

Inductive method_not_defined_in_class : CT -> methodName -> className -> Prop :=
  | mndic_obj : forall CT m,
      method_not_defined_in_class CT m Object
  | mndic_class : forall CT m C D Cf K M',
      lookup _ C CT = Some _ (class C D Cf K M')
    -> mlookup m M' = None _
    -> method_not_defined_in_class CT m D
    -> method_not_defined_in_class CT m C.

(* Valid method overriding *)
Inductive override : CT -> methodName -> className -> list className -> className -> Prop :=
   | m_notboundinsuper : forall CT m D C' C0,
        method_not_defined_in_class CT m D
     -> override CT m D C' C0
   | m_over : forall CT m D C' C0 D' D0,
        mtype CT m D D' D0
     -> C' = D'
     -> C0=D0
     -> override CT m D C' C0.

Fixpoint subst (x: list varName) (u: list tm)
               (C:className) (v:list tm)
               (t:tm) {struct t} : tm :=
  match t with
  | tm_var this => tm_new C v
  | tm_var y =>
      match lookup _ y (combine _ _ x u) with
        | Some u1 => u1
        | None => t
      end
  | tm_field t1 f =>
      tm_field (subst x u C v t1) f
  | tm_invoke t1 m l =>
      tm_invoke (subst x u C v t1) m
        ((fix subst_list (x: list varName) (u: list tm)
                         (C:className) (v:list tm) (tl: list tm)
                         {struct tl} : list tm :=
          match tl with
            | nil => nil _
            | (h::t) => (subst x u C v h) :: (subst_list x u C v t)
          end) x u C v l)
  | tm_new D l =>
      tm_new D ((fix subst_list (x: list varName) (u: list tm)
                                (C:className) (v:list tm) (tl: list tm)
                                {struct tl} : list tm :=
                match tl with
                | nil => nil _
                | (h::t) => (subst x u C v h) :: (subst_list x u C v t)
                end) x u C v l)
  | tm_cast D t1 => tm_cast D (subst x u C v t1)
  end.

Inductive eval : CT -> tm -> tm -> Prop :=
  | E_ProjNew : forall CT C v fj vj Cf,
         value_list v
      -> fields CT C Cf
      -> (lookup _ fj
           (combine _ _ (map _ _ (fun p => match p with (f,s) => f end) Cf) v))
          = Some _ vj
      -> eval CT (tm_field (tm_new C v) fj) (vj)
  | E_InvkNew : forall CT C v m u t0 x,
         value_list v
      -> value_list u
      -> mbody CT m C x t0
      -> eval CT (tm_invoke (tm_new C v) m u) (subst x u C v t0)
  | E_CastNew : forall CT C D v,
         value_list v
      -> subtyping CT C D
      -> eval CT (tm_cast D (tm_new C v)) (tm_new C v)
  | E_Field : forall CT f t t',
         eval CT t t'
      -> eval CT (tm_field t f) (tm_field t' f)
  | E_Invk_Recv : forall CT l m t0 t0',
         eval CT t0 t0'
      -> eval CT (tm_invoke t0 m l) (tm_invoke t0' m l)
  | E_Invk_Arg : forall CT v0 m v ti ti' t,
         value v0
      -> value_list v
      -> eval CT ti ti'
      -> eval CT (tm_invoke v0 m (v ++ [ti] ++ t))
                 (tm_invoke v0 m (v ++ [ti'] ++ t))
  | E_New_Arg : forall CT C v ti ti' t,
         value_list v
      -> eval CT ti ti'
      -> eval CT (tm_new C (v ++ [ti] ++ t)) (tm_new C (v ++ [ti'] ++ t))
  | E_Cast : forall CT C t t',
         eval CT t t'
      -> eval CT (tm_cast C t) (tm_cast C t').

Notation context := (alist className).

Definition empty : context := nil _.

Inductive typing : CT -> context -> tm -> className -> Prop :=
  | T_Var : forall CT Gamma x C,
         binds _ x C Gamma
      -> CT >> Gamma |- (tm_var x) ~ C
  | T_Field : forall CT Gamma t0 fi Ci C0 Cf,
         CT >> Gamma |- t0 ~ C0
      -> fields CT C0 Cf
      -> (lookup _ fi Cf) = Some _ Ci
      -> CT >> Gamma |- (tm_field t0 fi) ~ Ci
  | T_Invk : forall CT Gamma t0 m tl C C0 Dl,
         CT >> Gamma |- t0 ~ C0
      -> mtype CT m C0 Dl C
      -> typing_list CT Gamma tl Dl
      -> CT >> Gamma |- (tm_invoke t0 m tl) ~ C
  | T_New : forall CT Gamma tl C Df,
         fields CT C Df
      -> typing_list CT Gamma tl
            (map _ _ (fun p => match p with (f,s) => s end) Df)
      -> CT >> Gamma |- (tm_new C tl) ~ C
  | T_UCast : forall CT Gamma t0 C D,
         CT >> Gamma |- t0 ~ D
      -> subtyping CT D C
      -> CT >> Gamma |- (tm_cast C t0) ~ C
  | T_DCast : forall CT Gamma t0 C D,
           CT >> Gamma |- t0 ~ D
      -> subtyping CT C D
      -> C<>D
      -> CT >> Gamma |- (tm_cast C t0) ~ C
  | T_SCast : forall CT Gamma t0 C D,
           CT >> Gamma |- t0 ~ D
      -> ~ (subtyping CT C D)
      -> ~ (subtyping CT D C)
      -> CT >> Gamma |- (tm_cast C t0) ~ C

with typing_list : CT -> context -> list tm -> list className -> Prop :=
   | TL_nil  : forall CT Gamma,
              typing_list CT Gamma (nil _) (nil _)
   | TL_cons : forall CT Gamma t tl C Cl T,
              typing CT Gamma t T
         -> subtyping CT T C
 -> typing_list CT Gamma tl Cl
 -> typing_list CT Gamma (t::tl) (C::Cl)

where "CT >> Gamma |- t ~ T" := (typing CT Gamma t T).

(* Method typing *)
Inductive mtyping : CT -> className -> methodName
                 -> list (varName * className) -> tm ->className -> Prop :=
   | m_ok : forall CT C0 m Cx t0 C D Cl Df K Ml E0,
         CT >> (this,C) :: Cx |- t0 ~ E0
     -> subtyping CT E0 C0
     -> lookup _ C CT = Some _ (class C D Cf K Ml)
     -> override CT m D Cl C0
     -> mtyping CT C0 m Cx t0 C.
Hint Constructors mtyping.

Inductive mlist_typing : CT -> list M -> className -> Prop :=
   | m_ok_nil : forall CT C,
         mlist_typing CT (nil _) C
   | m_ok_cons : forall CT M C Ml C0 m Cx t0,
         M = method C0 m Cx t0
     -> mtyping CT C0 m Cx t0 C
     -> mlist_typing CT (M :: Ml) C.
Hint Constructors mlist_typing.

(* Class typing *)
```

```
Inductive ctyping : CT -> className -> className
                 -> list (fieldName * className) -> K -> list M -> Prop :=
   | c_ok : forall CT C D Cf K Ml Dg,
         K = constructor C (Dg ++ Cf)
               (map _ _ (fun p => match p with (f,s) => f end) Dg)
               (map _ _ (fun p => match p with (f,s) => f end) Cf)
     -> fields CT D Dg
     -> mlist_typing CT Ml C
     -> ctyping CT C D Cf K Ml.

Definition wf_class_table (ct : CT) : Prop :=
  forall C D Cf K M',
        lookup _ C ct = Some _ (class C D Cf K M')
     -> ctyping ct C D Cf K M'.
```