CIS 500 — Software Foundations

Final Exam

Review questions

December 11, 2007

The following problems should give you a general idea of the style and coverage of the exam questions from the last part of the course. Note that this is *not* a sample exam: The exam itself will cover all parts of the course, while the review sheet covers only new material since the second midterm.

Besides thinking about these problems, make sure that you can informally paraphrase the proofs of all the theorems and important technical lemmas for all of the languages we have studied since the second midterm, in the style of HW11. There *will* be a question on the exam that will require this background. (It will not necessarily be literally "Fill in an informal proof of the following theorem...," but having these informal proofs at your fingertips will make the question much easier to deal with.)

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 18 at the end.

1. Circle T or F for each of the following statements.

   (a) There exists a type that is a supertype of every other type.

   T F

   (b) There exists a type that is a subtype of every other type.

   T F

   (c) There exists a record type that is a subtype of every other record type.

   T F

   (d) There exists a record type that is a supertype of every other record type.

   T F

   (e) There exists an arrow type that is a subtype of every other arrow type.

   T F

   (f) There exists an arrow type that is a supertype of every other arrow type.

   T F

   (g) There is an infinite descending chain of distinct types in the subtype relation—that is, an infinite sequence of types $\mathtt{S}_0$, $\mathtt{S}_1$, etc., such that all the $\mathtt{S}_i$'s are different and each $\mathtt{S}_{i+1}$ is a subtype of $\mathtt{S}_i$.

   T F

   (h) There is an infinite ascending chain of distinct types in the subtype relation—that is, an infinite sequence of types $\mathtt{S}_0$, $\mathtt{S}_1$, etc., such that all the $\mathtt{S}_i$'s are different and each $\mathtt{S}_{i+1}$ is a supertype of $\mathtt{S}_i$.

   T F

2. What is the *smallest* type `T` ("smallest" in the subtype relation) that makes the following assertion true?

```
empty |- (\r~[[x~T]], r.x|]) @ [|x==(\z~A,z)|] ~ A-->A
```

`T =`

3. What is the *largest* type `T` that makes the same assertion true?

`T =`

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

```
empty |- (\r~[[x~A-->A]], [|y==(\z~B,z); r|]) @ [|x==(\z~A,z)|] ~ T
```

`T =`

5. What is the *largest* type `T` that makes the same assertion true?

`T =`

6. What is the *smallest* type `T` that makes the following assertion true?

```
[(a,A)] |- (\r~[[x~A; T]], r.y @ r.x) @ [|y==(\z~A,z), x==a|] ~ A
```

`T =`

7. What is the *largest* type `T` that makes the same assertion true?

`T =`

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

```
exists S, exists t,
  empty |- (\r~[[x~T]], r.x @ r.x) @ t ~ S
```

`T =`

9. 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 typing rule:

```
| T_Funny1 : forall Gamma t S1 S2 T1 T2,
       Gamma |- t ~ S1-->S2
    -> S1 <: S2
    -> S2 <: S1
    -> S2 <: T2
    -> Gamma |- t ~ T1-->T2
```

Progress: TRUE FALSE. For example...

Preservation: TRUE FALSE. For example...

(b) Suppose we add the following evaluation rule:

```
| E_Funny : forall x,
    -> eval [||] (\x~Top, x).
```

Progress: TRUE FALSE. For example...

Preservation: TRUE FALSE. For example...

(c) Suppose we add the following subtyping rule:

```
| S_Funny :
      [[]] <: Top-->Top
```

Progress: TRUE FALSE. For example...

Preservation: TRUE FALSE. For example...

(d) Suppose we add the following subtyping rule:

```
| S_Funny :
      Top-->Top <: []
```

Progress: TRUE FALSE. For example...

Preservation: TRUE FALSE. For example...

10. Recall the `S_Arrow` subtyping rule:

```
| S_Arrow : forall S1 S2 T1 T2,
      T1 <: S1
    -> S2 <: T2
    -> S1-->S2 <: T1-->T2
```

What happens to preservation and progress if we change this rule to the following?

```
| S_Arrow : forall S1 S2 T1 T2,
      S1 <: T1
    -> S2 <: T2
    -> S1-->S2 <: T1-->T2
```

11. In the file lec `2021.v` after the proof of Lemma `drop_duplicate_binding`, there is the following comment:

```
(* Note that we have to prove this by induction on typing derivations,
   not on terms as we did before.  OPTIONAL EXERCISE: Why? *)
```

Briefly explain.

12. Are the following statements true or false? (Circle T or F.)

(a)
```
(C-->C)-->[[x~A-->A,y~B-->B]]  <:  (C-->C)-->[[y~B-->B]]
```
T F

(b)
```
[[]] --> []]  <:  [[x~A]] --> Top
```
T F

(c)
```
forall S T,
    S <: T
 -> S-->S   <:  T-->T
```
T F

(d)
```
forall S T,
    S <: A-->A
 -> exists T,
      S = T-->T  /\  T <: A
```
T F

(e)
```
forall S T1 T1,
    S <: T1-->T2
 -> exists S1 S2,
      S = S1-->S2  /\  T1 <: S1  /\  S2 <: T2
```
T F

(f)
```
exists S,
    S <: S-->S
```
T F

(g)
```
exists S,
    S-->S <: S
```
T F

(h)
```
forall S T2 T2,
    S <: [[k~T1;T2]]
 -> exists S1 S2,
      S = [[k~S1;S2]]  /\  S1 <: T1  /\  S2 <: T2
```
T F

13. In Lecture 18, we defined *product types* and the associated term constructors *pairing, first projection, and second projection* as follows:

```
Inductive ty : Set :=
  ...
  | ty_prod     : ty -> ty -> ty.

Inductive tm : Set :=
  ...
  | tm_pair : tm -> tm -> tm
  | tm_proj1 : tm -> tm
  | tm_proj2 : tm -> tm.
```

We also introduced the notation `[[T1,T2]]` for `tm_prod T1 T2`.

Without looking back at Lecture 18, write down the typing and evaluation rules for products. Also, write down an appropriate subtyping rule for products.

Algorithmic Subtyping

14. The declarative subtyping relation `S <: T` defined on page 18 also cannot be translated "clause for clause" into an efficient recursive function that, given `S` and `T`, decides whether `S <: T`. Briefly explain why not.

Featherweight Java

There will be questions on the exam involving FJ. They will not demand familiarity with the proofs of FJ's properties, but make sure you understand the properties themselves (the statements of progress and preservation, in particular) and that you understand what the language is and have a basic understanding of how its formal definition works.

The definition of Featherweight Java is summarized on page 22 at the end.

15.
```
Inductive E : Set :=
   | ec_hole : E
   | ec_field : E -> fieldName -> E
   | ec_invk_recv : E -> methodName -> list tm -> E
   | ec_invk_arg : tm -> methodName -> list tm -> E -> list tm -> E
   | ec_new : className -> list tm -> E -> list tm -> E
   | ec_cast : className -> E -> E.

Fixpoint E_subst (e: E) (t: tm) {struct e} : tm :=
  match e with
   | ec_hole => t
   | ec_field c f => tm_field (E_subst c t) f
   | ec_invk_recv c m l => tm_invoke (E_subst c t) m l
   | ec_invk_arg v m vl c tl => tm_invoke v m (vl ++ [(E_subst c t)] ++ tl)
   | ec_new C vl c tl => tm_new C (vl ++ [(E_subst c t)] ++ tl)
   | ec_cast C c => tm_cast C (E_subst c t)
  end.

Theorem progress : forall CT Gamma t C,
     CT >> Gamma |- t ~ C
  -> (value t)
     \/ (exists t', eval t t')
     \/ (exists e:E, exists D:className, exists vl:list tm,
              (t = E_subst e (tm_cast C (tm_new D vl)))
           /\ (value_list vl)
           /\ (~ subtyping CT D C)).
```

Explain, in English, (1) why the ordinary formulation of progress is incorrect for FJ and (2) why this presentation of the progress theorem tells us something useful.

16. Recall the three typing rules for casts in FJ:

```
| 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
```

Why are there three rules instead of just one? Explain briefly.

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

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

Notation A := (ty_base one).
Notation B := (ty_base two).
Notation C := (ty_base three).
Notation " S --> T " := (ty_arrow S T) (at level 20, right associativity).

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

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

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, subst x s t1)
  | t1 @ t2 => (subst x s t1) @ (subst x s t2)
  end.

Notation "{ x |-> s } t" := (subst x s t) (at level 17).

Inductive value : tm -> Prop :=
  | v_abs : forall x T t,
      value (\x ~ T, t).

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').
```

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

Definition empty : context := nil _.

Reserved Notation "Gamma |- t ~ T" (at level 69).
Inductive typing : context -> tm -> ty -> Prop :=
  | T_Var : forall Gamma x T,
      binds _ x T Gamma ->
      Gamma |- !x ~ T
  | T_Abs : forall Gamma x T1 T2 t,
         (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
where "Gamma |- t ~ T" := (typing Gamma t T).
```

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'.
```