CIS 500 — Software Foundations
Midterm II

November 14, 2007

Name: ______________________________

Email: ______________________________

	Score
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
Total	

Instructions

- This is a closed-book exam: you may not use any books or notes.
- You have 80 minutes to answer all of the questions.
- Questions vary significantly in difficulty, and the point value of a given question is not always exactly proportional to its difficulty. Do not spend too much time on any one question.
- Partial credit will be given. All correct answers are short. The back side of each page may be used as a scratch pad.
- Good luck!

Typed arithmetic expressions

The full definition of the language of typed arithmetic and boolean expressions is reproduced, for your reference, on page 13.

1. (5 points) Answer "yes" or "no" for each part.

   (a) In this language, every normal form is a value.

   (b) In this language, every well-typed normal form is a value.

   (c) In this language, every value is a normal form.

   (d) In this language, the single-step evaluation relation is a partial function.

   (e) In this language, the single-step evaluation relation is a *total* function.

Untyped Lambda-Calculus

The following questions are about the untyped lambda calculus. For reference, the definition of this language and names for a number of specific lambda-terms (`c_zero`, `pls`, etc., etc.) appear on page 15 at the end of the exam.

2. (8 points) For each of the following terms, write down *how many* steps of single-step evaluation it takes for the term to reach a normal form. If the term is already a normal form, write "0". If the term has no normal form, write "diverges".

   (a) `fst @ (pair @ tru @ fls)`

   (b) `(\x, poisonpill) @ omega`

   (c) `pls @ c_one @ c_two`

   (d) `(\y, (\x, x @ x) @ (\x, x @ x) @ y) @ (\z, z)`

3. (3 points) Is the following statement true or false? If you write "false," give a counter-example.

   In the *pure* untyped lambda-calculus (i.e., the system we get by dropping constants from the untyped lambda-calculus summarized on page 15), every term either is a value or can take a step.

Programming in the Untyped Lambda-Calculus

Recall the definition of Church numerals and booleans in the untyped lambda-calculus (page 16).

4. (10 points) You may freely use the lambda-terms defined on page 16 in your answers to the following questions.

   (a) Write a lambda-term `swap` that reverses the elements of a pair. For example,

   ```
   fst @ (swap @ (pair @ AA @ BB))
   ```

   should evaluate to `BB`.

   (b) Write a lambda-term `minus` that subtracts Church numerals. For example `minus @ c_three @ c_one` should be behaviorally equivalent to `c_two`, and `minus @ c_one @ c_three` should be behaviorally equivalent to `c_zero`.

(c) Complete the following definition of a lambda-term `lt` that checks whether one Church numeral is strictly less than another. For example, `lt @ c_two @ c_three` should be behaviorally equivalent to `tru`, while `lt @ c_two @ c_two` and `lt @ c_two @ c_one` should be behaviorally equivalent to `fls`.

```
lt  =  Z  @  (\l, \m, \n,





           )
```

(This close parenthesis corresponds to the open parenthesis right after `Z` *above.)*

Behavioral Equivalence

Recall the definitions of observational and behavioral equivalence from the lecture notes:

- 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.

5. (2 points) Write "yes" or "no" for each of the following:

   (a) If two terms are behaviorally equivalent, then they are observationally equivalent.

   (b) If two terms are observationally equivalent, then they are behaviorally equivalent.

6. (8 points) Recall the hierarchy of program equivalences discussed in class:

   (a) syntactic equivalence (i.e., literal identity)
   (b) equivalence up to renaming of bound variables
   (c) equivalence modulo evaluation
   (d) behavioral equivalence
   (e) observational equivalence
   (f) universal equivalence (equating all terms)

   For each of the following pairs of terms, decide which is the *finest* equivalence (i.e., the one appearing earliest in the list) that relates the two terms, and write its letter. For example, if the two terms are behaviorally equivalent but not equivalent modulo evaluation, you would write "d".

   (a) `omega` and `poisonpill`

   (b) `omega @ tru` and `poisonpill @ tru`

   (c) `omega` and `poisonpill @ tru`

   (d) `tru` and `pls`

   (e) `fls` and `c_zero`

   (f)

```
Z @ (\f, \n,
        test
          @ (iszro @ n)
          @ (\z, c_one)
          @ (\z, tms @ n @ (f @ (prd @ n)))).
```

   and

```
Z @ (\f, \n,
        test
          @ (iszro @ n)
          @ (\z, c_one)
          @ (\z, f @ n)).
```

Simply Typed Lambda-Calculus

The following questions are about the simply typed lambda calculus. For reference, the definition of this language appears on page 17 at the end of the exam.

7. (10 points) Which of the following propositions are provable? Write "Yes" or "No" by each. For the ones where you write "Yes," give witnesses for the existentially bound variables. (E.g., for part 7a, give a type `T` such that `exists T, empty |- (\y:B-->B-->B, \x:B, y @ x) \in T` is provable.)

   (a) `exists T, empty |- (\y:B-->B-->B, \x:B, y @ x) \in T`

   (b) `exists T, empty |- (\x \in A-->B, \y \in B-->C, \z \in A, y @ (x @ z)) \in T`

   (c) `exists S, exists U, exists T, [(x,S), (y,U)]  |- (\z:A, x @ (y @ z)) \in T`

   (d) `exists S, exists T, [(x,S)] |- \y \in A, x @ (x @ y) \in T`

   (e) `exists S, exists U, exists T, [(x,S)] |- x @ (\z \in U, z @ x) \in T`

Recall that the simply typed lambda-calculus enjoys the following properties:

Determinacy (of single-step evaluation): if `eval t t'` and `eval t t''`, then `t' = t''`.

Progress: If `t` is closed and `empty |- t \in T`, then either `t` is a value or else there is some `t'` with `eval t t'`.

Preservation: If `empty |- t \in T` and `eval t t'`, then `t'` has type `T`.

8. (3 points) Suppose we add the following rule to the typing relation:

```
| T_Strange : forall x t,
      empty |- (\x \in A, t) \in B
```

Which of the three properties above become false in the presence of this rule? For each that becomes false, give a counter-example.

9. (3 points) Suppose we remove the rule `E_App1` from the evaluation relation. Which of the three properties above become false in the absence of this rule? For each that becomes false, give a counter-example.

10. (4 points) Here is the preservation theorem for the simply typed lambda-calculus:

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

Does the theorem remain true if we swap `t` and `t'` in the second premise (i.e., if we replace `eval t t'` by `eval t' t`)? Briefly explain.

11. (8 points) The following technical lemma plays a critical role in the proof of the preservation theorem:

```
Lemma substitution_preserves_typing : forall Gamma x U v t S,
     Gamma ++ [(x,U)] |- t \in S
  -> empty |- v \in U
  -> not_bound_in _ x Gamma
  -> Gamma |- {x|->v}t \in S.
```

(a) If we prove the preservation theorem by induction on evaluation derivations, there will be cases for the three evaluation rules, `E_AppAbs`, `E_App1` and `E_App2`. In which of these cases is the `substitution_preserves_typing` lemma used? (Just give the name of the case.)

(b) If instead we prove the preservation theorem by induction on typing derivations, there will be cases for the three typing rules, `T_Var`, `T_Abs` and `T_App`. In which of these cases is the lemma `substitution_preserves_typing` used? (Just give the name of the case.)

(c) What is the role of the typing context `Gamma` in the statement of `substitution_preserves_typing`? (Briefly explain.)

12. (14 points) Here is the progress theorem for the simply typed lambda-calculus:

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

Briefly fill in the blanks in the following informal outline of a proof of this theorem. (There are five blanks to fill in. Use English, not Coq! The correct answers are short.)

Proof. By induction on the derivation of `empty |- t \in T`.

- *Case `T_Var`:* Then `t = x`. ...

- *Case `T_Abs`:* Then `t = \x \in S, t1`. ...

- *Case `T_App`:* Then `t = t1 @ t2`. Since `t` is closed, so are `t1` and `t2`. By the induction hypothesis, we have

  ```
  value t1 \/ exists t1', eval t1 t1'
  ```

  and

  ```
  value t2 \/ exists t2', eval t2 t2'.
  ```

  Now:

  - If `t1` can take a step, then ...

  - If `t1` is a value and `t2` can take a step, then ...

  - If both are values, then ...

Challenge Problem

Warning: This problem is tricky and it is worth very few points. Do not attempt it until you have finished all the other questions and are confident of your answers.

13. (2 points) Let us call a term $\mathtt{t}_0$ in the pure untyped lambda-calculus *n-cyclic* if there exist n terms $\mathtt{t}_1$ ... $\mathtt{t}_n$ such that

    eval t $\mathtt{t}_1$
    eval $\mathtt{t}_1$ $\mathtt{t}_2$
    eval $\mathtt{t}_2$ $\mathtt{t}_3$
    ...
    eval $\mathtt{t}_{n-1}$ $\mathtt{t}_n$,

    where $\mathtt{t}_n = \mathtt{t}$ and $\mathtt{t}_i \neq \mathtt{t}$ for each $1 \leq i < n$. For example, `omega` is 1-cyclic.

    Prove that there exists an n-cyclic term for every $n \geq 1$.

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 \in 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 \in T, t1 => if eqnat x y then t else (\y \in 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 \in T, t).

Inductive eval : tm -> tm -> Prop :=
  | E_AppAbs : forall x T t12 v2,
         value v2
      -> eval ((\x \in 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 \in T" (at level 69).
Inductive typing : context -> tm -> ty -> Prop :=
  | T_Var : forall Gamma x T,
      binds _ x T Gamma ->
      Gamma |- !x \in T
  | T_Abs : forall Gamma x T1 T2 t,
         (x,T1) :: Gamma |- t \in T2
      -> Gamma |- (\x \in T1, t) \in T1-->T2
  | T_App : forall S T Gamma t1 t2,
         Gamma |- t1 \in S-->T
      -> Gamma |- t2 \in S
      -> Gamma |- t1@t2 \in T
where "Gamma |- t \in T" := (typing Gamma t T).
```