CIS 500 — Software Foundations

Midterm II

Review questions with answers

November 11, 2007

Work each of the review problems yourself before looking at the answers given here. If your answer differs from ours, make sure you understand why.

Typed arithmetic expressions

The full definition of the language of typed arithmetic and boolean expressions is reproduced, for your reference, on page 19. Here are some important properties enjoyed by this definition:

Determinacy (of single-step evaluation): if `eval t t′` and `eval t t″`, then `t′ = t″`.

Normalization (of many-step evaluation): For every term `t` there is some normal form `t′` such that `evalmany t t′`.

Progress: If `t` has type `T`, then either `t` is a value or else there is some `t′` with `eval t t′`.

Preservation: If `t` has type `T` and `eval t t′`, then `t′` has type `T`.

1. Suppose we add the following two new rules to the evaluation relation:

```
| E_PredTrue :
     eval (tm_pred tm_true)
          (tm_pred tm_false)

| E_PredFalse :
     eval (tm_pred tm_false)
          (tm_pred tm_true)
```

Which of the above properties will remain true in the presence of this rule? For each one, circle either "remains true" or else "becomes false." If a property becomes false, also write down a counter-example to the property.

(a) Determinacy of evaluation

remains true　　　　becomes false, because ....

Answer: Remains true

(b) Normalization

remains true　　　　becomes false, because ....

Answer: Becomes false: `pred true` $\longrightarrow$ `pred false` $\longrightarrow$ `pred true` *...*

(c) Progress

remains true　　　　becomes false, because ....

Answer: Remains true

(d) Preservation

remains true　　　　becomes false, because ....

Answer: Remains true

2. Suppose, instead, that we add this new rule to the typing relation:

```
| T_If : forall t2 t3,
       has_type t2 ty_nat
    -> has_type (tm_if tm_true t2 t3) ty_nat
```

Which of the properties on page 1 remains true? (Answer in the same style as the previous question.)

(a) Determinacy of evaluation

remains true   becomes false, because ....

Answer: Remains true

(b) Normalization

remains true   becomes false, because ....

Answer: Remains true

(c) Progress

remains true   becomes false, because ....

Answer: Remains true

(d) Preservation

remains true   becomes false, because ....

Answer: Remains true

3. Suppose, instead, that we add this new rule to the typing relation:

```
| T_SuccBool : forall t,
       has_type t ty_bool
    -> has_type (tm_succ t) ty_bool
```

Which of the properties on page 1 remains true? (Answer in the same style as the previous question.)

(a) Determinacy of evaluation

remains true | becomes false, because ....

Answer: Remains true

(b) Normalization

remains true | becomes false, because ....

Answer: Remains true

(c) Progress

remains true | becomes false, because ....

Answer: Becomes false: `(tm_succ tm_true)` *is well-typed, but stuck.*

(d) Preservation

remains true | becomes false, because ....

Answer: Remains true

4. Suppose we add a new rule

```
| E_Funny1 : forall t2 t3,
      eval (tm_if tm_true t2 t3)
           t3
```

to the ones given at the end of the exam. Do the properties on page 1 continue to hold in the presence of this rule?

For each property that *becomes false* when the proposed rule is added to the system, state the name of the property and give a brief counter-example demonstrating that it does not hold in the presence of the new rule.

Answer:

> **Determinism**: `tm_if tm_true tm_zero (tm_succ tm_zero)` *can now evaluate in one step to either* `tm_zero` *or* `(tm_succ tm_zero)`.

5. Suppose instead that we add this rule:

```
| E_Funny2 : forall t1 t2 t2' t3,
      eval t2 t2'
   -> eval (tm_if t1 t2 t3)
           (tm_if t1 t2' t3)
```

Answer in the same format as problem 4: For each property that becomes false when the proposed rule is added, write its name and give a brief counter-example. The properties are listed again at the bottom of this page for easy reference.

Answer:

> **Determinism**: `tm_if tm_false (tm_pred tm_zero) (tm_succ tm_zero)` *can now evaluate in one step to either* `tm_succ tm_zero` *or* `tm_if tm_false tm_zero (tm_succ tm_zero)`. *(There are several other correct counter-examples for this question.)*

6. Suppose instead that we add this rule to the original languge of typed arithmetic expressions:

```
| E_Funny3 :
      eval (tm_pred tm_false)
           (tm_pred (tm_pred tm_false))
```

Do the properties of the original system continue to hold in the presence of this rule?

Answer in the same format as the previous two problems.

Answer:

> **Normalization***: `tm_pred tm_false` diverges.*

7. Suppose instead that we add this rule to the original languge of typed arithmetic expressions:

```
| T_Funny4 :
      has_type tm_zero ty_bool
```

Do the properties of the original system continue to hold in the presence of this rule?

Answer in the same format as the previous three problems.

Answer:

> **Progress***: `tm_if tm_zero tm_true tm_true` has type `ty_bool`, is a normal form, and is not a value.*

8. Suppose instead that we add this rule to the original languge of typed arithmetic expressions:

```
| T_Funny5 :
      has_type (tm_pred tm_zero) ty_bool
```

Do the properties of the original system continue to hold in the presence of this rule?

Answer in the same format as the previous problems.

Answer:

> **Preservation***: `tm_pred tm_zero` has type `ty_bool` and evaluates in one step to `tm_zero`, which does not have type `ty_bool`.*

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 21 at the end of the exam.

9. Circle the term that each of the following lambda calculus terms steps to, using the single-step evaluation relation `eval t t'`. If the term is a normal form, circle DOESN'T STEP.

   (a) `(\x,x) @ (\x, x @ x) @ (\x, x @ x)`

      i. `(\x, x) @ (\x, x @ x) @ (\x, x @ x)`

      ii. `(\x, x @ x) @ (\x, x @ x)`

      iii. `(\x', (\x, x @ x)) @ (\x, x @ x)`

      iv. `(\x, x) @ (\x, x @ x)`

      v. DOESN'T STEP

   Answer: (ii)

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

      i. `(\x, (\x,x) @ (\x, x @ x))`

      ii. `(\x, (\x, x @ x))`

      iii. `(\x, (\x, x))`

      iv. `(\x, x) @ (\x, x @ x)`

      v. DOESN'T STEP

   Answer: (v)

   (c) `(\x, (\z, \x, x @ z) @ x) @ (\x, x @ x)`

      i. `(\x, (\z, \x, x @ z) @ x) @ (\x, x @ x)`

      ii. `(\z, \x', (\x, x @ x) @ z) @ (\x, x @ x)`

      iii. `(\z, \x, x @ z) @ (\x, x @ x)`

      iv. `(\x, x @ (\x, x @ x))`

      v. DOESN'T STEP

   Answer: (iii)

10. For each of the following terms, either write down the term that it steps to in a single step of evaluation or else write "DOESN'T STEP" if the term is a normal form.

    (a) `(\x, \y, x @ (\x, y @ x) @ y) @ (\z, z @ y)`
    Answer: `\y, (\z, z @ y) @ (\x, y @ x) @ y`

    (b) `\z, (\x, \y, x @ x) @ (\x, \y, x @ x)`
    Answer: Doesn't step

11. Circle the normal forms of the following lambda calculus terms, if one exists. If there is no normal form, circle NONE.

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

       i. `(\y, (\z, z @ z) @ y) @ (\x, x)`
       ii. `(\z, z @ z) @ (\x,x)`
       iii. `(\x, x)`
       iv. `(\y, y @ y) @ (\z,z)`
       v. NONE

       Answer: iii

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

       i. `(\x, x @ x @ x) @ (\x, x @ x @ x)`
       ii. `(\x, x @ x @ x)`
       iii. `(\x, x @ x @ x) @ (\x, x @ x @ x) @ (\x, x @ x @ x)`
       iv. `x @ x @ x`
       v. NONE

       Answer: v

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

       i. `(\x, (\y, y @ y) @ (\z, z @ z))`
       ii. `(\x, (\y, y @ y))`
       iii. `(\y, (\z, z @ z))`
       iv. `(\y, y @ y) @ (\z, z @ z)`
       v. NONE

       Answer: i

Programming in the Untyped Lambda-Calculus

Recall the definition of Church numerals and booleans in the untyped lambda-calculus (page 22).

12. Which of these lambda calculus terms implements `xor` (the exclusive or function, which returns `tru` when exactly one of its arguments is `tru`.)

    (a) `\x, \y, x @ (y @ fls @ tru) @ (y @ tru @ fls)`

    (b) `\x, \y, x @ y @ y`

    (c) `\x, \y, tru @ x @ y`

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

    Answer: (a)

13. Which of these lambda-calculus terms implements `odd`, a function that returns `tru` if its argument (the encoding of a natural number) is odd and `fls` otherwise,

    (a) `\m, m @ (\n, n @ fls @ tru) @ fls`

    (b) `\m, m @ fls @ (\n, tru @ fls)`

    (c) `\m, fls @ (\n, n @ m @ tru)`

    (d) `\m, m @ (\n, tru) @ fls`

    Answer: (a)

14. The following is a slightly different encoding of natural numbers in the untyped lambda calculus.

```
s_0 = \s,\z, z
s_1 = \s,\z, s @ s_0 @ z
s_2 = \s,\z, s @ s_1 @ (s @ s_0 @ z)
s_3 = \s,\z, s @ s_2 @ (s @ s_1 @ (s @ s_0 @ z))

scc = \n,\s,\z, s @ n @ (n @ s @ z)
```

    (a) Define the predecessor function `prd` for this encoding, using the simplest term you can.
    Answer: `prd = \n, n @ (\m,\r, m) @ s_0`

    (b) Define the addition function `pls` for this encoding, using the simplest term you can.
    Answer: `pls = \n, \m, n @ (\x, scc) @ m`
    or the same definition for `pls` *as for Church numerals:*
    `pls = \n, \m, \s,\z, n @ s @ (m @ s @ z)`

    (c) Define the function `sumupto` that, given the encoding of a number `m`, calculates the sum of all the numbers less than or equal to `m`. Use the simplest term you can, and do not use `Z`.
    Answer: `sumupto = \m, m @ plus @ m` *is the simplest answer.*

15. Complete the following definition of a lambda-term `equal` that implements a *recursive* equality function on Church numerals. For example, `equal @ c_zero @ c_zero` and `equal @ c_two @ c_two` should be behaviorally equivalent to `tru`, while `equal @ c_zero @ c_one` and `equal @ c_three @ c_zero` should be behaviorally equivalent to `fls`. You may freely use the lambda-terms defined on page 22.

```
equal =
 Z @ (\e,
       \m, \n,
         test @ (iszro @ m)
ANSWER:
           @ (\dummy, (iszro @ n))
           @ (\dummy,
              test @ (not @ (iszro @ n))
                @ (\dummy, e @ (prd @ m) @ (prd @ n))
                @ (\dummy, fls)))
```

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.

16. For each of the following pairs of terms, write *Yes* if the terms are behaviorally equivalent and *No* if they are not.

(a)
```
plus @ c_2 @ c_1
c_3
```
Answer: Yes

(b)
```
\x, \y, x @ y
\x, x
```
Answer: Yes

(c)
```
tru
\x, \y, (\z, z) @ x
```
Answer: Yes

(d)
```
\x, \y, x @ y
\x, \y, (\z, z) @ x @ y
```
Answer: Yes

(e)
```
\x, \y, x @ y
\x, \y, x @ (\z, z) @ y
```
Answer: No

(f)
```
(\x, x @ x) @ (\x, x @ x)
(\x, x @ x @ x) @ (\x, x @ x @ x)
```
Answer: Yes

(g)
```
(\x, x @ x) @ (\x, x @ x)
\x, (\x, x @ x) @ (\x, x @ x)
```
Answer: No

(h)
```
\x, \y, x @ y
\x, x
```
Answer: Yes

(i)
```
\f, (\x, f @ (x @ x)) @ (\x, f @ (x @ x))
(\f, (\y, (\x, f @ (\y, x @ x @ y)) @ (\x, f @ (\y, x @ x @ y)) @ y))
```
Answer: No

In the following problems, feel free to use the lambda-terms (`c_zero`, `omega`, etc., etc.) defined on page 22.

17. The terms `tru` and `fls` are not behaviorally equivalent. Show this by writing down a list `[v_1, v_2, ..., v_n]` of closed values such that

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

and

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

are *not* observationally equivalent. (Give the shortest possible such list.)

Answer: *[tru, poisonpill, tru]*

18. The terms `omega` and `poisonpill` are not behaviorally equivalent. Show this by writing down a list `[v_1, v_2, ..., v_n]` of closed values such that

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

and

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

are *not* observationally equivalent. (Give the shortest possible such list.)

Answer: *[ ] (the empty list)*

19. The terms `c_two` and `c_three` are not behaviorally equivalent. Show this by writing down a list `[v_1, v_2, ..., v_n]` of closed values such that

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

and

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

are *not* observationally equivalent. (Give the shortest possible such list.)

Answer: *[(\x,\x,\x,omega), tru]*

Alternative Notions of Evaluation

20. One attractive feature of behavioral equivalence is that the definitions can be applied verbatim to other notions of evaluation besides standard call-by-value evaluation. In this problem, we'll apply them to call-by-name (CBN) evaluation.

    Recall the definition of single-step CBN evaluation from Lecture 15:

```
Inductive eval_cbn : tm -> tm -> Prop :=
  | En_AppAbs : forall x t12 v2,
        eval_cbn ((\x, t12) @ v2) ({x |-> v2} t12)
  | En_App1 : forall t1 t1' t2,
        eval_cbn t1 t1'
      -> eval_cbn (t1 @ t2) (t1' @ t2).
```

- Two terms `s` and `t` are *observationally equivalent under CBN* iff either both are normalizable (i.e., they reach a normal form after a finite number of CBN evaluation steps) or both are divergent.
- Terms `s` and `t` are *behaviorally equivalent under CBN* 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 under CBN.

For each of the following pairs of terms, write *Yes* if the terms are behaviorally equivalent under CBN and *No* if they are not.

(a) `omega`
`tru`
Answer: No

(b) `omega`
`poisonpill`
Answer: No

(c) `tru @ c_zero @ omega`
`tru @ c_zero @ fls`
Answer: Yes

(d) `tru @ omega @ c_zero`
`tru @ fls @ c_zero`
Answer: Yes

(e) `\x, \y, x @ y`
`\x, x`
Answer: Yes

(f) `tru`
`\x, \y, (\z, z) @ x`
Answer: Yes

(g) `tru`
`\x, \y, (\z, z) @ x`
Answer: Yes

(h) `(\x, x @ x) @ (\x, x @ x)`
`\x, (\x, x @ x) @ (\x, x @ x)`
Answer: No

Simply Typed Lambda-Calculus

The following questions are about the untyped lambda calculus. For reference, the definition of this language appears on page 23 at the end of the exam.

21. Which of the following propositions are provable? (Write "Yes" or "No" by each.)

    (a) `[(y,B)] |- (\x \in A, x) \in A-->A`
    Answer: Yes

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

    (c) `exists T, empty |- (\y:B-->B, \x:B, x @ y) \in T`
    Answer: No

    (d) `exists S, exists T, [(x,S)] |- x @ (\y:B-->B, y) \in T`
    Answer: Yes

    (e) `exists S, [(x,S)] |- (\y:B-->B, y) @ x \in S`
    Answer: Yes

    (f) `exists S, exists T, [(x,S)] |- x @ x @ x \in T`
    Answer: No

22. State the progress and preservation theorems for the simply typed lambda-calculus (without looking at the lecture notes).

Answer:

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

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

(* Or, equivalently: *)
Theorem progress : forall t T,
     empty |- t \in T
  ->    value t
     \/ exists t', eval t t'.
```

23. The following technical lemma appears in the notes for Lecture 17:

```
Lemma weakening_empty_preserves_typing : forall Gamma t T,
     empty |- t \in T
  -> Gamma |- t \in T.
```

Briefly explain where this property is used and why it is needed.

Answer: It is used in the proof that substitution preserves typing. When we encounter an occurrence of the variable being substituted for, we need to replace it with the term `v` *being substituted in and use the proof that* `v` *is well typed (with the same type as we are assuming for the variable). But this proof is given in the empty context, whereas we are using it in a setting where we've gone under some binders and the context may be non-empty. The* `weakening_empty_preserves_typing` *lemma is used to show that the typing proof for* `v` *can be "weakened" to apply in this non-empty context.*

Note: It is very likely that there will be at least one question on the exam where you will be expected to remember how the most important properties of the untyped and/or simply typed lambda-calculus are proved (progress, preservation, determinism, weakening, substitution, etc.).

However, the phrasing of this question is more challenging than would probably be used on an exam, in the sense that it gives you almost no guidance as to what sort of response is desired (how much detail to give, what you can assume the reader remembers, how formal to be, etc.). An exam question would be somewhat more structured.

Coq Tactics

24. Briefly explain what the following tactics do:

   (a) `subst`

   Answer: The `subst]` tactic eliminates (from the context) all *equalities where one side is just a variable by rewriting all occurrences of this variable in the goal and all the other hypotheses. This is a good way to clean up a mess left by `inversion`.*

   (b) `try solve [t1 | t2 | ...]`

   Answer: `try solve [t1 | t2 | ...]` will try to solve the goal by using first tactic `t1`, then `t2`, etc. If none of them succeeds in completely solving the goal, then `try solve [t1 | t2 | ...]` does nothing.

   (c) `t1 ; t2`

   Answer: Applies `t1` to the current goal and then applies `t2` to every subgoal *generated by `t1`.*

   (d) `assumption`

   Answer: If the context contains an assumption `H` that will solve the goal and generate no subgoals, then doing `assumption` is just the same as doing `apply H` (except that `H` does not need to be named explicitly).

   (e) `remember`

   Answer: `remember e as x` replaces all occurrences of the expression `e` (in the current goal and in the current context) with the variable `x` and introduces an assumption `x = e`.

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: Untyped 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).
```