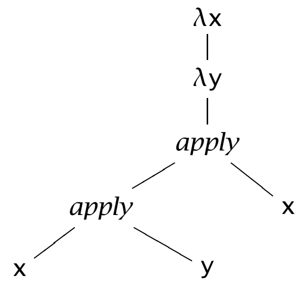
CIS 500 — Software Foundations

Midterm I, Review Questions
With answers

Untyped lambda-calculus

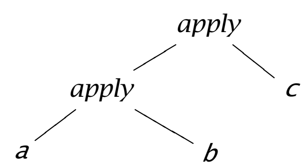
1. (2 points) We have seen that a linear expression like `λx. λy. x y x` is shorthand for an abstract syntax tree that can be drawn like this:



Draw the abstract syntax trees corresponding to the following expressions:

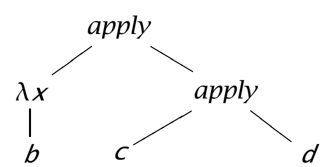
(a) `a b c`

Answer:



(b) `(λx. b) (c d)`

Answer:



2. (10 points) Write down the normal forms of the following λ-terms:

(a) `(λt. λf. t) (λt. λf. f) (λx. x)`

Answer: *λt. λf. f*

(b) `(λx. x) (λx. x) (λx. x) (λx. x)`

Answer: *λx. x*

(c) `λx. x (λx. x) (λx. x)`

Answer: *λx. x (λx. x) (λx. x)*

(d) `(λx. x (λx. x)) (λx. x (λx. x x))`

Answer: *λx. x x*

(e) `(λx. x x x) (λx. x x x)`

Answer: No normal form

3. (4 points) Recall the following abbreviations from Chapter 5:

```
tru = λt. λf. t
fls = λt. λf. f
not = λb. b fls tru
```

Complete this definition of a lambda term that takes two church booleans, `b` and `c`, and returns the logical "exclusive or" of `b` and `c`.

```
xor = λb. λc. ______________________________________________
```

Some possible answers:

```
xor = λb. λc. b (not c) c
xor = λb. λc. b (c fls tru) c
```

4. (8 points) A list can be represented in the lambda-calculus by its `fold` function. (OCaml's name for this function is `fold_right`; it is also sometimes called `reduce`.) For example, the list `[x,y,z]` becomes a function that takes two arguments `c` and `n` and returns `c x (c y (c z n))`. The definitions of `nil` and `cons` for this representation of lists are as follows:

```
nil = λc. λn. n;
cons = λh. λt. λc. λn. c h (t c n);
```

Suppose we now want to define a λ-term `append` that, when applied to two lists `l1` and `l2`, will append `l1` to `l2` — i.e., it will return a λ-term representing a list containing all the elements of `l1` and then those of `l2`. Complete the following definition of `append`.

```
append = λl1. λl2. λc. λn. ____________________________________________________________
```

Answer:

```
append = λl1. λl2. λc. λn. l1 c (l2 c n)
```

5. (6 points) Recall the call-by-value fixed-point combinator from Chapter 5:

```
fix = λf. (λx. f (λy. x x y)) (λx. f (λy. x x y));
```

We can use `fix` to write a function `sumupto` that, given a Church numerals `m`, calculates the sum of all the numbers less than or equal to `m`, as follows.

```
g = λf. λm.
      (iszro m)
         (λx. c₀)
         (λx. plus _________ (_________ (prd m)))
      tru;
sumupto = fix g;
```

Fill in the two omitted subterms.
Answer:

```
g = λf. λm.
      (iszro m)
         (λx. c₀)
         (λx. plus m (f (prd m)))
      tru;
```

Nameless representation of terms

6. (4 points) Suppose we have defined the naming context $\Gamma = \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}$. What are the deBruijn representations of the following λ-terms?

   (a) λx. λy. x y d

   Answer: λ. λ. *1 0 2*

   (b) λx. c (λy. (c y) x) d

   Answer: λ. *2 (*λ*. (3 0) 1) 1*

7. (4 points) Write down (in deBruijn notation) the terms that result from the following substitutions.

   (a) $[0 \mapsto \lambda.0]((\lambda.\ 0\ 1)\ 1)$

   Answer: (λ*. 0 (*λ*.0)) 1*

   (b) $[0 \mapsto \lambda.\ 0\ 1]((\lambda.\ 0\ 1)\ 0)$

   Answer: (λ*. 0 (*λ*. 0 2)) (*λ*. 0 1)*

Typed arithmetic expressions

The full definition of the language of typed arithmetic and boolean expressions is reproduced, for your reference, on page 10.

8. (9 points) Suppose we add the following new rule to the evaluation relation:

$$\texttt{succ true} \longrightarrow \texttt{pred (succ true)}$$

Which of the following properties will remain true in the presence of this rule? For each one, write either "remains true" or else "becomes false," plus (in either case) a one-sentence justification of your answer.

(a) Termination of evaluation (for every term $\texttt{t}$ there is some normal form $\texttt{t}'$ such that $\texttt{t} \longrightarrow^* \texttt{t}'$)

Answer: Becomes false. For example, the term succ true has no normal form.

(b) Progress (if $\texttt{t}$ is well typed, then either $\texttt{t}$ is a value or else $\texttt{t} \longrightarrow \texttt{t}'$ for some $\texttt{t}'$)

Answer: Remains true. Adding a new evaluation rule can only make it easier for the progress property to hold.

(c) Preservation (if $\texttt{t}$ has type $\texttt{T}$ and $\texttt{t} \longrightarrow \texttt{t}'$, then $\texttt{t}'$ also has type $\texttt{T}$)

Answer: Remains true: succ true is not well typed (nor is any term containing it), so it doesn't matter what it evaluates to.

9. (9 points) Suppose, instead, that we add this new rule to the evaluation relation:

$$\texttt{t} \longrightarrow \texttt{if true then t else succ false}$$

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

(a) Termination of evaluation (for every term $\texttt{t}$ there is some normal form $\texttt{t}'$ such that $\texttt{t} \longrightarrow^* \texttt{t}'$)

Answer: Becomes false. For any term t, we can evaluate $t \longrightarrow$ *if true then t else succ false* $\longrightarrow t \longrightarrow \ldots$

(b) Progress (if $\texttt{t}$ is well typed, then either $\texttt{t}$ is a value or else $\texttt{t} \longrightarrow \texttt{t}'$ for some $\texttt{t}'$)

Answer: Remains true. As above, adding a new evaluation rule can only make it easier for the progress property to hold.

(c) Preservation (if $\texttt{t}$ has type $\texttt{T}$ and $\texttt{t} \longrightarrow \texttt{t}'$, then $\texttt{t}'$ also has type $\texttt{T}$)

Answer: Becomes false: a well typed term like zero can now evaluate to the ill-typed term if true then zero else succ false.

10. (9 points) Suppose, instead, that we add a new type, `Funny`, and add this new rule to the typing relation:

$$\texttt{if true then false else false} : \texttt{Funny}$$

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

(a) Termination of evaluation (for every term $\texttt{t}$ there is some normal form $\texttt{t}'$ such that $\texttt{t} \longrightarrow^* \texttt{t}'$)
Answer: Remains true. Adding typing rules doesn't change the evaluation relation or its properties.

(b) Progress (if $\texttt{t}$ is well typed, then either $\texttt{t}$ is a value or else $\texttt{t} \longrightarrow \texttt{t}'$ for some $\texttt{t}'$)
Answer: Remains true. This rule doesn't make any new terms well typed.

(c) Preservation (if $\texttt{t}$ has type $\texttt{T}$ and $\texttt{t} \longrightarrow \texttt{t}'$, then $\texttt{t}'$ also has type $\texttt{T}$)
Answer: Becomes false: for example, the term `if true then false else false` has type `Funny`, but reduces to `false`, which does not have type `Funny`.

Simply typed lambda-calculus

The definition of the simply typed lambda-calculus with booleans is reproduced for your reference on page 12.

11. (6 points) Write down the types of each of the following terms (or "ill typed" if the term has no type).

    (a) `λx:Bool. x x`
    Answer: ill typed

    (b) `λf: Bool→Bool. λg:Bool→Bool. g (f (g true))`
    Answer: (Bool→Bool)→(Bool→Bool)→Bool

    (c) `λh:Bool. (λi:Bool→Bool. i false) (λk:Bool.true)`
    Answer: Bool→Bool

12. (9 points) Recall the rules for "big-step evaluation" of arithmetic and boolean expressions from HW 3.

$$\mathtt{v} \Downarrow \mathtt{v}$$

$$\frac{\mathtt{t}_1 \Downarrow \mathtt{true} \qquad \mathtt{t}_2 \Downarrow \mathtt{v}_2}{\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \Downarrow \mathtt{v}_2}$$

$$\frac{\mathtt{t}_1 \Downarrow \mathtt{false} \qquad \mathtt{t}_3 \Downarrow \mathtt{v}_3}{\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \Downarrow \mathtt{v}_3}$$

$$\frac{\mathtt{t}_1 \Downarrow \mathtt{nv}_1}{\mathtt{succ}\ \mathtt{t}_1 \Downarrow \mathtt{succ}\ \mathtt{nv}_1}$$

$$\frac{\mathtt{t}_1 \Downarrow 0}{\mathtt{pred}\ \mathtt{t}_1 \Downarrow 0}$$

$$\frac{\mathtt{t}_1 \Downarrow \mathtt{succ}\ \mathtt{nv}_1}{\mathtt{pred}\ \mathtt{t}_1 \Downarrow \mathtt{nv}_1}$$

$$\frac{\mathtt{t}_1 \Downarrow 0}{\mathtt{iszero}\ \mathtt{t}_1 \Downarrow \mathtt{true}}$$

$$\frac{\mathtt{t}_1 \Downarrow \mathtt{succ}\ \mathtt{nv}_1}{\mathtt{iszero}\ \mathtt{t}_1 \Downarrow \mathtt{false}}$$

The following OCaml definitions implement this evaluation relation *almost correctly*, but there are three mistakes in the `eval` function—one each in the `TmIf`, `TmSucc`, and `TmPred` cases of the outer match. Show how to change the code to repair these mistakes. (Hint: all the mistakes are *omissions*.)

```
let rec isnumericval t = match t with
    TmZero(_) → true
  | TmSucc(_,t1) → isnumericval t1
  | _ → false

let rec isval t = match t with
    TmTrue(_)  → true
  | TmFalse(_) → true
  | t when isnumericval t  → true
  | _ → false

let rec eval t = match t with
    v when isval v → v
  | TmIf(_,t1,t2,t3) →
      (match t1 with
          TmTrue _ → eval t2
        | TmFalse _ → eval t3
        | _ → raise NoRuleApplies)
  | TmSucc(fi,t1) →
      (match eval t1 with
          nv1 → TmSucc (dummyinfo, nv1)
        | _ → raise NoRuleApplies)
  | TmPred(fi,t1) →
      (match eval t1 with
          TmZero _ → TmZero(dummyinfo)
        | _ → raise NoRuleApplies)
  | TmIsZero(fi,t1) →
      (match eval t1 with
          TmZero _ → TmTrue(dummyinfo)
        | TmSucc(_, _) → TmFalse(dummyinfo)
        | _ → raise NoRuleApplies)
  | _ → raise NoRuleApplies
```

Answer:

- *In the `TmIf` clause, `match t1 with` should be `match (eval t1) with`.*
- *In the `TmSucc` clause, the guard `nv1 → ...` should be `nv1 when isnumericval nv1 → ...` — or, equivalently, the body of the clause, `TmSucc (dummyinfo, nv1)`, should be replaced by `if isnumericval nv1 then TmSucc (dummyinfo, nv1) else raise NoRuleApplies`*
- *In the `TmPred` clause, the whole case*

  ```
  | TmSucc(_, nv1) → nv1
  ```

  is missing from the inner `match` (it should follow the `TmZero` case).