CIS 624, Fall 2015, Final Examination
7 December 2015

Please do not turn the page until everyone is ready.

Rules:

• The exam is closed-book, notes are allowed.

• Please stop promptly at 16:45.

• You can rip apart the pages, but please write your name on each page.

• There are 151 points total, distributed unevenly among 6 questions. A perfect score is 100 points, any extra points over 100 will be added to your midterm score (if it was less than 100).

• Most questions have multiple parts. You will receive points for any parts you complete.

• You are not expected to complete all questions.

Advice:

• Read questions carefully. Understand a question before you start writing.

• Write down thoughts and intermediate steps so you can get partial credit.

• The questions are not necessarily in order of difficulty. Skip questions you are not confident about and if you have time, come back to them later. Remember you only need 100 points total.

• If you have questions, ask.

• Relax. You are here to learn.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Max points</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>100 (+51)</strong></td>
<td></td>
</tr>
</tbody>
</table>
IMP Language

\[ s ::= \text{skip} \mid x := e \mid s \mid \text{if } e \text{ } s \mid \text{while } e \text{ } s \]
\[ e ::= c \mid x \mid e + e \mid e \times e \]
\[(c \in \{-2, -1, 0, 1, 2, \ldots\})\]
\[(x \in \{x_1, x_2, \ldots, y_1, y_2, \ldots, z_1, z_2, \ldots, \ldots\})\]

\[ H ; e \downarrow c \]

\[ \text{CONST} \quad \text{VAR} \quad \text{ADD} \]
\[ H ; c \downarrow c \quad H ; x \downarrow H(x) \quad H ; e_1 + e_2 \downarrow c_1 + c_2 \]
\[ H ; e_1 \downarrow c_1 \quad H ; e_2 \downarrow c_2 \quad H ; e_1 \times e_2 \downarrow c_1 \times c_2 \]

\[ H_1 ; s_1 \rightarrow H_2 ; s_2 \]

\[ \text{ASSIGN} \quad \text{SEQ1} \quad \text{SEQ2} \]
\[ H ; e \downarrow c \quad H ; \text{skip} ; s \rightarrow H ; s \quad H ; s_1 ; s_2 \rightarrow H' ; s_1' ; s_2 \]
\[ H ; x := e \rightarrow H ; x \mapsto c \quad \text{skip} \quad \text{skip} \]
\[ \text{IF1} \quad \text{IF2} \quad \text{WHILE} \]
\[ H ; e \downarrow c \quad \text{if } e_1 \quad \text{while } e_1 \]
\[ c > 0 \quad \text{if } e \text{ } s_1 \text{ } s_2 \rightarrow H ; s_1 \quad \text{if } e \text{ } s_1 \text{ } s_2 \rightarrow H ; s_2 \]
\[ \text{WHILE} \]
\[ H ; \text{while } e \text{ } s \rightarrow H ; \text{if } e \text{ } (s \text{ } \text{while } e \text{ } s) \text{ } \text{skip} \]

Simply Typed Lambda Calculus with constants

\[ e ::= \lambda x. e \mid x \mid e \mid e \mid c \]
\[ v ::= \lambda x. e \mid c \]
\[ \tau ::= \text{int} \mid \tau \rightarrow \tau \]

\[ e \rightarrow e' \text{ and } \Gamma \vdash e : \tau \text{ and } \tau_1 \leq \tau_2 \]

\[ (\lambda x. e) v \rightarrow e[v/x] \quad e_1 \rightarrow e'_1 \quad e_2 \rightarrow e'_2 \]
\[ e_1 e_2 \rightarrow e'_1 e_2 \quad v e_2 \rightarrow v e'_2 \]

\[ \Gamma \vdash c : \text{int} \quad \Gamma \vdash x : \Gamma(x) \quad \Gamma, x : \tau_1 \vdash e : \tau_2 \quad \Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2 \]
\[ \Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \]
\[ \Gamma \vdash e_1 e_2 : \tau_1 \]

\[ \tau_3 \leq \tau_1 \quad \tau_2 \leq \tau_4 \quad (\text{S-Arrow}) \quad \tau \leq \tau \quad (\text{S-Refl}) \quad \tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3 \quad (\text{S-Trans}) \]
- If $\cdot \vdash e : \tau$ and $e \rightarrow e'$, then $\cdot \vdash e' : \tau$.
- If $\cdot \vdash e : \tau$, then $e$ is a value or there exists an $e'$ such that $e \rightarrow e'$.
- If $\Gamma, x : \tau' \vdash e : \tau$ and $\Gamma \vdash e' : \tau'$, then $\Gamma \vdash e[e'/x] : \tau$.

**System F (syntax)**

| $e$ | ::= | $c | x | \lambda x : \tau. e | e e | \Lambda \alpha. e | e[\tau]$ | $\Gamma$ | ::= | $\cdot | \Gamma, x : \tau$ |
| $\tau$ | ::= | $\text{int} | \tau \rightarrow \tau | \alpha | \forall \alpha. \tau$ | $\Delta$ | ::= | $\cdot | \Delta, \alpha$ |

$\nu e e$

| $\nu e e$ | ::= | $\lambda x : \tau. e | e \alpha. e$ |

**System F: $e \rightarrow e'$ and $\Delta; \Gamma \vdash e : \tau$**

| $e \rightarrow e'$ | $\Delta; \Gamma \vdash e \rightarrow e'$ | $e \rightarrow e'$ |
| $e e_2 \rightarrow e' e_2$ | $e \rightarrow e'$ | $(\nu e \tau)e \rightarrow (\nu e \tau)[e/x]$ |
| $\nu e e \rightarrow \nu e e'$ | $(\nu e \tau) \rightarrow (\nu e \tau)[\nu e \tau]$ | $(\nu e \tau)[\nu e \tau] \rightarrow (\nu e \tau)[\nu e \tau][\nu e \tau]$ |

**Simple System F examples:** Let $\text{id} = \Lambda \alpha. \lambda x : \alpha. x$. Then $\text{id}$ has type $\forall \alpha. \alpha \rightarrow \alpha$; $\text{id}[\text{int}]$ has type $\text{int} \rightarrow \text{int}$; and $\text{id}[\text{int} * \text{int}]$ has type $(\text{int} * \text{int}) \rightarrow (\text{int} * \text{int})$.

**Sum types, iso-recurvise types**

| $e$ | ::= | $\ldots | A(e) | B(e) | (\text{match } e \text{ with } A x. e | B x. e) | \text{fold}_\tau e | \text{unfold} e$ |
| $\tau$ | ::= | $\ldots | \tau_1 + \tau_2 | \mu \alpha. \tau$ |
| $v$ | ::= | $\ldots | A(v) | B(v) | \text{fold}_\tau v$ |

**match $A(v)$ with $A x. e_1 | B y. e_2 \rightarrow e_1[v/x]$**

| $e \rightarrow e'$ | $e \rightarrow e'$ | $e \rightarrow e'$ |
| $A(e) \rightarrow A(e')$ | $B(e) \rightarrow B(e')$ | match $e$ with $A x. e_1 | B y. e_2 \rightarrow$ match $e'$ with $A x. e_1 | B y. e_2$ |

**match $B(v)$ with $A x. e_1 | B y. e_2 \rightarrow e_2[v/y]$**

| $e \rightarrow e'$ | $e \rightarrow e'$ | $e \rightarrow e'$ |
| unfold $(\text{fold}_{\mu \alpha. \tau} v) \rightarrow v$ | $\text{fold}_{\mu \alpha. \tau} e \rightarrow \text{fold}_{\mu \alpha. \tau} e'$ | $\text{unfold} e \rightarrow \text{unfold} e'$ |

**match $e$ with $A x. e_1 | B y. e_2 \rightarrow$ match $e'$ with $A x. e_1 | B y. e_2$**

| $\Delta; \Gamma \vdash e : \tau_1 + \tau_2$ | $\Delta; \Gamma \vdash e : \tau_1 + \tau_2$ | $\Delta; \Gamma \vdash e : \tau[\mu \alpha. \tau] / \alpha$ |

**match $e$ with $A x. e_1 | B y. e_2 : \tau$**

| $\Delta; \Gamma \vdash e : \tau_1$ | $\Delta; \Gamma \vdash e : \tau_2$ | $\Delta; \Gamma \vdash e : \tau[\mu \alpha. \tau] / \alpha$ |

| $\Delta; \Gamma \vdash \text{fold}_{\mu \alpha. \tau} e : \mu \alpha. \tau$ | $\Delta; \Gamma \vdash \text{unfold} e : \tau[\mu \alpha. \tau] / \alpha$ | $\Delta; \Gamma \vdash \text{fold}_{\mu \alpha. \tau} e : \mu \alpha. \tau$ |
1. (22 points) OCaml

(a) (10 points) Describe what, if anything, each of the following OCaml programs would print:

i. let f x y = x y in
   let z = f print_string "hi " in
   f print_string "hi"

ii. let f x = (fun y -> print_string x) in
    let g = f "elves " in
    let x = "trees " in
    g "cookies "

iii. let rec f n x =
      if n>=0
      then (let _ = print_string x in f (n-1) x)
      else ()
      in
      f 3 "hi "

iv. let rec f n x =
    if n>=0
    then (let _ = print_string x in f (n-1) x)
    else ()
    in
    f 3

v. let rec f x = f x in
   print_string (f "hi ")
(b) (12 points) Implement a recursive function in OCaml that computes the average of a list of integers. You can use integer division. Do not traverse the list more than once. What is the type of the function you implemented?

Solution:

(a) i. hi hi
   ii. elves
   iii. hi hi hi hi
   iv. prints nothing (evaluates to a function that prints when called)
   v. prints nothing (goes into an infinite loop)

(b) let rec avg l =
    let rec f l =
    match l with
    [] -> (0,0)
    | h::tl ->
      let n, s = f tl in
      n + 1, h + s
    in let length, sum = f l in sum / length;;

The type is int list -> int
2. (34 points) IMP. In this problem, we consider an expression language that is like expressions in IMP except we remove multiplication and we add a *global counter*, which is initialized to 0 by the interpreter. Our new syntax for expressions is:

\[ e ::= c \mid x \mid e + e \mid \text{next} \]

The syntax for statements remains unchanged. Informally, the *next* expression evaluates to the current counter value and has the side-effect of incrementing the counter value by 1. For example, if the current counter value is 3, evaluating *next* returns 3 and then changes the internal counter value to 4.

(a) (9 points) For each of the following IMP programs, answer these questions: will the computation terminate and if yes, what will the value of \( x \) be at the end?

i. \( x := 1 + \text{next} + \text{next} \)

ii. \( y := 2; \text{while } y \ (x := 1 + \text{next}; y := y + (-1)) \)

iii. \( x := 1 + \text{next}; \text{while } x \ (x := \text{next} + (-1)) \)

(b) (8 points) Define large-step semantics rules for this expression language. The judgment should have the form \( H; c_1; e \Downarrow c_2; c \) where:

- \( H, e, \) and \( c \) are like in IMP.
- \( c_1 \) is the value of the global counter before evaluation.
- \( c_2 \) is the value of the global counter after evaluation.
(c) (12 points) Prove this theorem: If $H; c_1; e \downarrow c_2; c$ and $c'_1 > c_1$, then there exist $c'_2$ and $c'$ such that $H; c'_1; e \downarrow c'_2; c'$ and $c'_2 > c_2$.

(d) (5 points) Suppose we also extend IMP statement semantics to support the global counter (so the judgment has the form $H; c; s \rightarrow H'; c'; s'$). Argue that this theorem is false: If $H_1; c_1; s \rightarrow^* H_2; c_2; \textbf{skip}$ and $c'_1 > c_1$, then there exist $H'_2$ and $c'_2$ such that $H_1; c'_1; s \rightarrow^* H'_2; c'_2; \textbf{skip}$ and $c'_2 > c_2$. You do not need to give the semantic rules for statements or show a full state sequence. Just give an example showing the theorem is false and explain why informally.

Solution:

(a)  
  i. yes, x is 2
  ii. yes, x is 2
  iii. yes, x is 0
(b) 

<table>
<thead>
<tr>
<th>CONSTR</th>
<th>VAR</th>
<th>ADD</th>
</tr>
</thead>
<tbody>
<tr>
<td>( H; c_1; c_2 \downarrow c_1; c_2 )</td>
<td>( H; c_1; x \downarrow c_1; H(x) )</td>
<td>( H; c_1; e_1 \downarrow c_1; c_1 )</td>
</tr>
<tr>
<td>( H; c_1; next \downarrow c_1; c_1 )</td>
<td>( H; c_1; x \downarrow c_1; H(x) )</td>
<td>( H; c_1; x + e_2 \downarrow c''; c_1 + c_2 )</td>
</tr>
</tbody>
</table>

(c) By induction on the derivation of \( H; c_1; e \downarrow c_2; c \):

- If the derivation ends with \textsc{const}, then \( c_2 = c_1 \) and we can use \textsc{const} to derive \( H; c_1; e \downarrow c_1; c \). Since \( c_1' > c_1 = c_2 \), letting \( c_2' = c_1' \) (and \( c' = c \)) suffices.
- If the derivation ends with \textsc{var}, then \( c_2 = c_1 \), and we can use \textsc{var} to derive \( H; c_1 \downarrow c_1; c \). Since \( c_1' > c_1 = c_2 \), letting \( c_2' = c_1' \) (and \( c' = c \)) suffices.
- If the derivation ends with \textsc{add}, then \( e = e_1 + e_2 \) and there exists some \( c_3, c_4, \) and \( c_5 \) such that \( H; c_1; e_1 \downarrow c_3; c_4 \) and \( H; c_3; e_2 \downarrow c_2; c_5 \). So by induction on the derivation for \( e_1 \) there exist \( c_3' > c_3 \) and \( c_4' \) such that \( H; c_1' \downarrow c_3'; c_4' \). Since \( c_3' > c_3 \), by induction on the derivation for \( e_2 \) there exist \( c_2' > c_2 \) and \( c_5' \) such that \( H; c_3' \downarrow c_2'; c_5' \). So using \textsc{add} with \( H; c_1'; e_1 \downarrow c_3'; c_4' \) and \( H; c_3'; e_2 \downarrow c_2'; c_5' \) we can derive \( H; c_1'; e_1 + e_2 \downarrow c_2'; c_4' + c_5' \) where \( c_2' > c_2 \).
- If the derivation ends with \textsc{next}, then \( c_2 = c_1 + 1 \) and we can use \textsc{next} to derive \( H; c_1; \text{next} \downarrow c_1 + 1; c_1 \). Since \( c_1' > c_1 \), we know \( c_1' + 1 > c_1 + 1 = c_2 \).

(d) The essence of the problem is conditionals (or loops). For example, consider \( s = \) \text{if next skip next}. If \( c_1 = 0 \) and \( c_1' = 1 \), then \( H; c_1; s \rightarrow^* H; 2; \text{skip} \) and \( H; c_1'; s \rightarrow^* H; 2; \text{skip} \), but \( 2 \neq 2 \).
3. (30 points) Coin-flipping in Lambda-Calculus

In this problem we take the simply-typed lambda-calculus with conditionals (assume that lambda abstractions for the following have been defined: \texttt{true}, \texttt{false}, \texttt{if} \ e_1 \ e_2 \ e_3, and the type \texttt{bool}). For example, the following (curried) function returns the exclusive-or of its arguments: \( \lambda x. \lambda y. \text{if } x \ (\text{if } y \ \text{false} \ x) \ y. \)

Add a “coin-flip” expression to the language using the keyword \texttt{flip}. This expression is not a value. Our call-by-value left-to-right small-step semantics has two new semantic rules:

\[
\text{flip} \to \text{true} \quad \text{flip} \to \text{false}
\]

(a) (5 points) Argue that for all \( e \), \( (\lambda x. \ e) \text{true} \) and \( e[\text{true}/x] \) are equivalent under call-by-value.

(b) (8 points) Argue that depending on \( e \), \( (\lambda x. \ e) \text{flip} \) and \( e[\text{flip}/x] \) may or may not be equivalent under call-by-value.

(c) (5 points) Give a typing rule for \texttt{flip}.

(d) (12 points) Assuming we have proofs of progress, preservation, and substitution for lambda-calculus with conditionals, explain how to extend the proofs for programs containing \texttt{flip}. Be clear about the induction hypothesis and what cases you are adding.
(a) Given \((\lambda x. e) \text{ true}\), only one evaluation rule applies and produces \(e[\text{true}/x]\). So the expressions can produce exactly the same results (or non-termination); \((\lambda x. e) \text{ true}\) just takes one more step.

(b) An example where they are equivalent is \(e = x\); both expressions always terminate and evaluate to \text{true} or \text{false}. An example where they are not equivalent is when \(\lambda x. e\) is our “xor” function from part (a). If we apply “xor” to \text{flip} under call-by-value we get either a function equivalent to negation (if \text{flip} \rightarrow \text{true}) or the identity function (if \text{flip} \rightarrow \text{false}). But the body of “xor” with \text{flip} substituted for \(x\) is a function that when passed \text{false} could return \text{true} or \text{false}. (There are simpler examples; the point is \(e\) needs to use \(x\) more than once.)

(c) \[ \Gamma \vdash \text{flip} : \text{bool} \]

(d) • Preservation: We show if \(\cdot \vdash e : \tau\) and \(e \rightarrow e'\), then \(\cdot \vdash e' : \tau\) by induction on the derivation of \(\cdot \vdash e : \tau\). The one new case is when \(e\) is \text{flip}, so \(\tau\) is \text{bool}. By inspecting the operational semantics, two rules apply and \(e'\) is \text{true} or \text{false}. In either case, we have axioms that let us derive \(\cdot \vdash \text{true} : \text{bool}\) and \(\cdot \vdash \text{false} : \text{bool}\).

• Progress: We show if \(\cdot \vdash e : \tau\), then \(e\) is a value or there exists an \(e'\) such that \(e \rightarrow e'\) by induction on the derivation of \(\cdot \vdash e : \tau\). The one new case is when \(e\) is \text{flip}. In this case, we can always take a step, for example let \(e'\) be \text{true}.

• Substitution: We show if \(\Gamma, x: \tau' \vdash e : \tau\) and \(\Gamma \vdash e' : \tau'\), then \(\Gamma \vdash e[e'/x] : \tau\) by induction on the derivation of \(\Gamma, x: \tau' \vdash e : \tau\). The one new case is when \(e\) is \text{flip} so \(\tau\) is \text{bool}. In this case, \(e[e'/x]\) is also \text{flip} and we can derive \(\Gamma \vdash \text{flip} : \text{bool}\).
4. (20 points)

(a) (12 points) Consider this System F function. Note the comma in it, which creates a pair. (We also assume System F has pairs and strings.)

\[
\lambda x : \text{int}. \lambda y : \text{string}. \lambda z : \forall \alpha. \forall \beta. (\alpha \rightarrow \beta \rightarrow \alpha). ((z \ [\text{string}] \ [\text{int}] \ y \ x), (z \ [\text{int}] \ [\text{string}] \ x \ y))
\]

i. What does this function do? Be as precise as possible.

ii. Why is it not possible to write a function equivalent to this one in OCaml?

(b) (8 points) Consider this OCaml function.

\[
\text{let rec } f \ x \ y = \text{if } x < y \text{ then } (x,y) \text{ else } f \ y \ x
\]

i. What does this function do? Be as precise as possible.

ii. Why is it not possible to write a function equivalent to this one in System F?
Solution:

(a) This function takes three arguments $x$, $y$, and $z$ and always returns the pair $(y, x)$. (Note that $z$ must be a function that always terminates and always returns its first argument.)

This function cannot be implemented in OCaml for two reasons. First, it cannot type-check because $z$ is used at two different types. Second, there is no way to require the caller to pass in a function that takes two arguments and always returns the first one.

(b) This function takes two integers $x$ and $y$. It returns $(x, y)$ if $x$ is less than $y$, $(y, x)$ if $y$ is less than $x$, and diverges if $x = y$. In System F, no function can ever diverge.
5. (25 points) Continuation passing style in OCaml.

Assume that the following functions are defined as follows.

```ocaml
open List
let eqk a b k = k (a = b)
let addk a b k = k (a + b)
let subk a b k = k (a - b)
let times a b k = k (a * b)
let divk a b k = k (a / b)
let hdk lst k = k (hd lst)
let tlk lst k = k (tl lst)
let print_int i =
    print_string (string_of_int i); print_newline();
```

Using only the above functions, implement the following CPS functions.

(a) (10 points) Write a CPS-style function that takes a list of integers and a continuation function as arguments and computes the sum of all the elements in the list. For example `sumk [1;2;3;4;8] print_int;;` should output 18.
(b) (10 points) Write a CPS-style function that takes a list of integers and a continuation function as arguments and computes the length of the list.
For example `countk [1;2;3;4;8] print_int;;` should output 5.

(c) (5 points) Write a CPS-style function that takes a list of integers and a continuation function as arguments and computes the average of all elements in the list. Use the provided functions and the `sumk` and `countk` functions you implemented above.
For example `avgk [1;2;3;4;8] print_int;;` should output 3.

Solution:

(a) let rec sumk l k =
    eqk l []
    (fun empty -> if empty then k 0
    else hdk l
    (fun h -> tlk l
    (fun ltail -> sumk ltail
    (fun tailsum -> addk h tailsum k))));

(b) let rec countk l k =
    eqk l []
    (fun empty -> if empty then k 0
    else tlk l
    (fun ltail -> countk ltail
    (fun lsize -> addk 1 lsize k)));

(c) let rec avgk l k =
    countk l (fun lsize -> sumk l
    (fun lsum -> divk lsum lsize k));
6. (20 points)
Consider a typed λ-calculus with sum types, pair types, recursive types, unit, string and int.

(a) Define a type \( t_1 \) (starting with \( \mu \alpha. \)) for a binary tree where:
   • Each interior node has no data and two children.
   • Each leaf node has either a \texttt{string} or an \texttt{int}.

(b) Give a type \( t_2 \) (starting with \( \mu \alpha. \)) for a binary tree where:
   • Each internal node has no data and two \textit{optional} children (meaning each child may
     or may not be another binary tree).
   • Each leaf node is either a \texttt{string} or has no data.

(c) Give an example tree of type \( t_1 \) that cannot be translated to an equivalent value of type \( t_2 \).

(d) Give an example tree of type \( t_2 \) that cannot be translated to an equivalent value of type \( t_1 \).

Solution:

(a) \( t_1 = \mu \alpha. (\texttt{int} + \texttt{string}) + (\alpha \times \alpha) \)

(b) \( t_2 = \mu \alpha. (\texttt{unit} + \texttt{string}) + (\texttt{unit} + \alpha) \times (\texttt{unit} + \alpha) \)

(c) A tree with one node containing an \texttt{int}, e.g., 3. Many other correct answers possible.

(d) A tree with one \texttt{unit} node (no data). Many other correct answers possible.