CIS 624, Fall 2014, Final Examination
9 December 2014

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 **165 points** total, distributed *unevenly* among 7 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>12</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>100 (+65)</strong></td>
<td></td>
</tr>
</tbody>
</table>
Simply Typed Lambda Calculus with pairs

\[
\begin{align*}
\text{let } e := \lambda x. e \mid x := e \mid s \mid \text{if } e \text{ s} \mid \text{while } e \text{ s} \\
\text{let } e := c \mid x \mid e + e \mid e \ast e \\
(c \in \{ \ldots, -2, -1, 0, 1, 2, \ldots \}) \\
(x \in \{ x_1, x_2, \ldots, y_1, y_2, \ldots, z_1, z_2, \ldots, \ldots \})
\end{align*}
\]

**IMP Language**

\[
\begin{align*}
\text{assign } & H \\
\text{seq1 } & \Gamma \vdash e, \gamma \\
\text{seq2 } & H ; s_1 \rightarrow H' ; s_2 \\
\text{while } & H ; \text{while } e \text{ s} \rightarrow H ; \text{if } (s; \text{while } e \text{ s}) \text{ skip}
\end{align*}
\]

**Simply Typed Lambda Calculus with pairs**

\[
\begin{align*}
e & := \lambda x. e \mid x \mid e e \mid c \mid (e, e) \mid e.1 \mid e.2 \\
v & := \lambda x. e \mid c \mid (v, v) \\
\tau & := \text{int} \mid \tau \rightarrow \tau \mid \tau \ast \tau
\end{align*}
\]

\[
\begin{array}{cccc}
e \rightarrow e' \text{ and } \Gamma \vdash e : \tau \text{ and } \tau_1 \leq \tau_2 & \hline \\
\lambda x. e \rightarrow e'[v/x] & e_1 \rightarrow e'_1 & e_2 \rightarrow e'_2 & (v_1, e_2) \rightarrow (v_1, e_2) \\
& e_1 \rightarrow e'_1 & e_2 \rightarrow e'_2 & (v_1, e_2).1 \rightarrow v_1 \\
& e.1 \rightarrow e'.1 & e.2 \rightarrow e'.2 & (v_1, v_2).2 \rightarrow v_2 \\
\end{array}
\]

\[
\begin{align*}
\Gamma \vdash c : \text{int} & \\
\Gamma \vdash x : \Gamma(x) & \\
\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2 & \\
\Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 & \\
\Gamma \vdash e_2 : \tau_1 & \\
\end{align*}
\]

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

**System F (syntax)**

$e ::= c \mid x \mid \lambda \alpha. e \mid e \mid \Lambda \alpha. e \mid e[\tau]$

$\tau ::= \text{int} \mid \tau \to \tau \mid \alpha \mid \forall \alpha. \tau$

$v ::= c \mid \lambda \alpha. e \mid \Lambda \alpha. e$

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

$\frac{e \to e'}{\Delta; \Gamma \vdash e \to e'}$

$\frac{e \to e'}{\Delta; \Gamma \vdash e \to e'}$

$\frac{(\lambda x : \tau. e)v \to e[v/x]}{\Delta; \Gamma \vdash (\Lambda \alpha. e)[\tau] \to e[\tau/\alpha]}$

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

**Sum types, iso-recursive types**

$e ::= \ldots \mid A(e) \mid B(e) \mid (\text{match } e \text{ with } A. e \mid B. e) \mid \text{fold}_\tau e \mid \text{unfold } e$

$\tau :: = \ldots \mid \tau_1 + \tau_2 \mid \mu \alpha. \tau$

$v ::= \ldots \mid A(v) \mid B(v) \mid \text{fold}_\tau v$

match $A(v)$ with $A. e_1 \mid B. e_2 \to e_1[v/x]$

match $B(v)$ with $A. e_1 \mid B. e_2 \to e_2[v/y]$
1. (12 points) For each of the following OCaml definitions, does it type-check in OCaml? If so, what type does it have? If not, why not?

(a) let a = 3 in (fun f -> (fun x y -> x) (f a) (f true))

(b) let b = (fun f -> (fun x y -> x) (f 1) (f (f (f 5))))

(c) let c = (fun x y z -> x y z) (fun p q -> p * q) 5 10

(d) let d = (fun f -> (fun x y -> y) (f 3) (f (-10)))

Solution:

(a) Does not type-check: The type-inferencer will conclude that g must be a function takes an int and a function that takes a bool, and these cannot both hold.

(b) Type-checks: (int -> int) -> int

(c) Type-checks: int

(d) Type-checks: (int -> 'a) -> 'a
2. (30 points) We want to extend IMP (defined on p. 2) with case conditional of the form

\[
\text{case } e \text{ of }
\]
\[
c1 \text{ : } s1;
\]
\[
c2 \text{ : } s2;
\]
\[
\ldots;
\]
\[
cn \text{ : } sn
\]
\[
\text{endcase}
\]

where \textit{case, of, and encase} are new keywords; \( e \) is an arithmetic expression, each \( c_i \) is an integer constant, and each \( s_i \) is a statement. This program is executed by first evaluating the expression \( e \) to obtain a constant \( c \); if the first occurrence of \( c \) in the list \( c_1, \ldots, c_n \) is \( c_i \) (duplicates are allowed in the list \( c_1, \ldots, c_n \)), then the statement \( s_i \) is executed. If \( c \) does not occur in the list \( c_1, \ldots, c_n \), then the program immediately terminates (i.e., is equivalent to \textit{skip}).

(a) (10 points) Give a BNF definition of the syntax of case conditionals by extending the current IMP definition of statements, \( s \). It can be helpful to (optionally) use a separate metavariable \textit{CaseList} for the list of cases between \textit{of} and \textit{endcase}.

\[
\text{CaseList} ::= c : s \mid c : s ; \text{CaseList}
\]
\[
s ::= \text{skip} \mid x := e \mid s ; s \mid \text{if } e s s \mid \text{while } e s
\]

(b) (10 points) Give small-step operational semantics for case statements (you should have at least two new rules).
(c) (10 points) What is the value of x at the end of the program below? Show a correct sequence of steps, specifying the small-step judgement rule(s) used in each step.

```plaintext
x := 3;
case 2 * x of
  3: x := -1;
  6: x := x + (-1);
  5: x := x + 1;
  6: x := 0
endcase
```

Solution:

(a)  
\[
\text{CaseList ::= } c : s \mid c : s; \text{CaseList}
\]

\[
s ::= \text{skip} \mid x ::= e \mid s ; s \mid \text{if } e \text{ } s \text{ } s \mid \text{while } e \text{ } s
\]

| case e of CaseList endcase

(b)  
\[
\text{CASE1}
\]

\[
H ; e \Downarrow c_i\quad \text{\rightarrow}\quad H ; \text{case } e \text{ of } c_1 : s_1 ; \cdots ; c_i : s_i ; \cdots ; c_n : s_n \text{ endcase } \rightarrow H ; s_i
\]

\[
\text{CASE2}
\]

\[
H ; e \Downarrow c\quad c \notin \{c_1, \cdots, c_n\}
\quad \text{\rightarrow}\quad H ; \text{case } e \text{ of } c_1 : s_1 ; \cdots ; c_i : s_i ; \cdots ; c_n : s_n \text{ endcase } \rightarrow H ; \text{skip}
\]

(c)  
\[
H = \{\}; x := 3; \text{case } 2 * x \text{ of } 3 : x := -1; 6 : x := x - 1; 5 : x := x + 1; 6 : x := 0 \text{ endcase}
\rightarrow^2 H = x \rightarrow 3; \text{case } 2 * x \text{ of } 3 : x := -1; 6 : x := x - 1; 5 : x := x + 1; 6 : x := 0 \text{ endcase}
\]

\[
[\text{Seq2, Assign}]
\]

\[
\rightarrow H = x \rightarrow 3; x := x - 1 \quad [\text{Case1}]
\]

\[
\rightarrow H = x \rightarrow 3, x \rightarrow 2; \text{skip} \quad [\text{Assign}]
\]

7
3. (20 points) Define a list encoding using the simply-typed lambda calculus with functions, and integers as considered in class. A non-empty list can be represented as $\lambda s. s \; h \; t$ where $h$ and $t$ are the head and tail of the list.

You can (optionally) use the definition of booleans and pairs from lecture, or other helper expressions.

```
"true"  \lambda x. \lambda y. x
"false" \lambda x. \lambda y. y
"mkpair" \lambda x. \lambda y. \lambda z. z \; x \; y
"fst"   \lambda p. \lambda x. \lambda y. x
"snd"   \lambda p. \lambda x. \lambda y. y
```

Define the lambda functions for each of the following operations. You can use previously defined shortcut names (e.g., “mkpair”, “true”, etc.).

(a) Create an empty list:

(b) Check if a list is empty:

(c) Create a non-empty list containing the integers 1, 2, and 3 (you can use numbers directly, no need for Church encoding).

(d) Get the last element (tail) of the list containing 1, 2, and 3 (you can use numbers directly, no need for Church encoding).

Solution:
(a) “emptylst” = “mkpair” “false” “false” (each node represented by a pair whose first element is head of the list, and second element is the tail; “false” as the first element of a pair designates the empty list)

(b) “isempty” = “fst” = \( \lambda p. p(\lambda x. \lambda y. x) = \lambda z. \lambda p. p(\lambda p. p(\lambda x. \lambda y. y) z) \)

(c) “mkpair” “1” (“mkpair” “2” (“mkpair” “3” “false”))

(d) “tail” = \( \lambda z. \ “snd” \ (“snd” (“snd” z)) = \lambda z. \lambda p. p(\lambda p. p(\lambda x. \lambda y. y) z) \)
4. (26 points) This problem uses System F with pairs and extended with integer pairs and a new operation, pair subtraction. For example, with pair subtraction \((3, 4) - (1, 3)\) should result in \((2, 1)\). Note that the answers to all parts should be brief.

(a) Define a large-step operational rule for subtraction of expressions of the form \(e_1 - e_2\) where \(e_1\) and \(e_2\) can be reduced to values that are pairs of integer constants.

\[
\text{E-SUB} \\
\frac{}{e_1 - e_2 \Downarrow}
\]

(b) Give the appropriate System F typing rule for subtraction of expressions of the form \(e_1 - e_2\) where the types of \(e_1\) and \(e_2\) are pairs of ints.

\[
\text{T-SUB} \\
\frac{}{\Delta; \Gamma \vdash e_1 - e_2 :}
\]

(c) Consider a typing context where:

- There are no type variables in scope.
- \(x\) is the only term variable in scope and it has type \(\forall \alpha. \alpha \rightarrow \alpha\). **Correction:** The type should have been \(\forall \alpha. \alpha \rightarrow \alpha \rightarrow \alpha\); answers satisfying either type will be accepted without any point deduction.

i. What does \(\tau\) need to be for the program fragment

\[
x \ [\tau] \ (\lambda y : \text{int} \ast \text{int.} \ \lambda z : \text{int} \ast \text{int.} \ y - z) \ (10, 2) \ (5, 2)
\]

to typecheck? (Recall application — of types or terms — associates to the left.)

\(\tau\) is

ii. Given your choice for \(\tau\) above, what is the type of

\[
x \ [\tau] \ (\lambda y : \text{int} \ast \text{int.} \ \lambda z : \text{int} \ast \text{int.} \ y - z) \ (10, 2) \ (5, 2)
\]
(d) If $v$ is an arbitrary value such that

$$v \ [\tau] \ (\lambda y : \text{int} \ast \text{int} . \ \lambda z : \text{int} \ast \text{int} . \ y - z) \ (10, 2) \ (5, 2)$$

type-checks (notice $v$ is a *value* and no longer polymorphic), then:

i. What type does $v$ have? (Hint: it’s different from the answers to part c).

ii. What might the following expression evaluate to?

$$v \ (\lambda y : \text{int} \ast \text{int} . \ \lambda z : \text{int} \ast \text{int} . \ y - z) \ (10, 2) \ (5, 2)$$

Solution:

(a)

$$
\frac{e_1 \downarrow (c_1, c_2) \quad e_2 \downarrow (c_3, c_4)}{e_1 - e_2 \downarrow (c_1 - c_3, c_2 - c_4)}
$$

(b)

$$
\frac{\Delta; \Gamma \vdash e_1 : \text{int} \ast \text{int} \quad \Delta; \Gamma \vdash e_2 : \text{int} \ast \text{int}}{\Delta; \Gamma \vdash e_1 - e_2 : \text{int} \ast \text{int}}
$$

(c)  

i. $\tau$ must be $\text{int} \ast \text{int} \to \text{int} \ast \text{int} \to \text{int} \ast \text{int}$
ii. $\text{int} \ast \text{int}$

(d)  

i. $(\text{int} \ast \text{int} \to \text{int} \ast \text{int} \to \text{int} \ast \text{int}) \to (\text{int} \ast \text{int} \to \tau_1)$ for any $\tau_1$.
ii. It could produce any value whatsoever.
5. (15 points)
Consider a typed λ-calculus with sum types, pair types, recursive types, unit, and int.

(a) Define a type $t_1$ for a binary tree of integers where:
   - Each interior node has one integer and two children.
   - Each leaf node has no data.
   - Your type definition should have the form $\mu \alpha \cdots$.

(b) Give a type $t_2$ for a binary tree of integers where:
   - Each node has one integer and two optional children (meaning each child may or may not be another binary tree).
   - Your type definition should have the form $\mu \alpha \cdots$.

(c) Explain in English how there is exactly one value of type $t_1$ that cannot be translated to an equivalent value of type $t_2$.

Solution:

(a) $\mu \alpha. \text{unit} + (\text{int} \times \alpha \times \alpha)$
(b) $\mu \alpha. \text{int} \times (\text{unit} + \alpha) \times (\text{unit} + \alpha)$
(c) The empty tree can be represented with a value of type $t_1$ but not with $t_2$ because every $t_2$ has at least one int.
6. Continuation passing style in OCaml.

(a) (12 points) Assume that the `eqk`, `addk`, `timesk`, `divk` functions are defined as follows.

```ocaml
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);;
```

Using only the above functions, implement a CPS function `abcdk` that takes four integer arguments `a, b, c, d`, a regular continuation `k`, and an exception continuation `xk`, to compute the following integer expression: \( a \times (b + c)/d \). If `d` is 0, call the exception continuation `xk` and pass the offending value to it.

```ocaml
# let abcdk a b c d k xk = ...;;
val abcdk : int -> int -> int -> int -> (int -> 'a) -> (int -> 'a) -> 'a = <fun>
```
(b) (20 points) Consider the direct style function that given a list of integers, returns the sum of squares of all values.

\[
\text{let rec sumsquares } l = \\
\text{ match } l \text{ with} \\
\text{ [] } \rightarrow \text{ 0} \\
\text{ | h::t } \rightarrow \text{ (h*h) + (sumsquares t)}
\]

i. What is the type of \text{sumsquares} above?

ii. For a given call to \text{sumsquares} above, approximately how deep would the call-stack grow in terms of the function arguments?

iii. Write a version of \text{sumsquares} called \text{sumsquaresk} in continuation-passing style (i.e., it should take as arguments a list of integers and a continuation function:
\[
\text{let rec sumsquaresk } l \text{ k } = \ldots,
\]
which uses a small constant amount of stack space. You can assume that the following CPS functions are defined (you can assume only integer division is supported, e.g., \text{divk \ 5 \ 2 (fun x->x) returns 2}).

```ocaml
open List;;
let eqk arg1 arg2 k = k (arg1 == arg2);;
let timesk arg1 arg2 k = k (arg1 * arg2);;
let divk arg1 arg2 k = k (arg1 / arg2);;
let hdk lst k = k (hd lst);;
let tlk lst k = k (tl lst);;
let plusk arg1 arg2 k = k (arg1 + arg2);;
```

(continues on next page)
iv. What is the type of the `sumsquaresk` function you wrote in part b.iii?

Solution:

(a) \(\frac{a \times (b + c)}{d}\)

```ocaml
let abcdk a b c d k xk =
  eqk d 0
  (fun ex -> if ex then xk d
      else plusk b c
        (fun bc -> timesk a bc
          (fun abc -> divk abc d k))));;
```

(b) Sum the squares of values in list.

i. `int list -> int`

ii. Its depth will be proportional to the length of the list `l`.

iii. let rec sumsquaresk l k =
    eqk l []
    (fun empty -> if empty then k 0
        else hdk l
            (fun h -> timesk h h
              (fun h2 -> tlk l
                (fun ltail -> sumsquaresk ltail
                  (fun t -> plusk h2 t k))))));;

(* To test: *)

```ocaml
let print_int i =
  print_string (string_of_int i); print_newline();;

sumsquaresk [1;2;3] print_int;;
```

iv. `sumsquaresk` has type `int list -> (int -> 'a) -> 'a`
7. (30 points) In this problem, we consider a call-by-value lambda-calculus with very basic support for profiling: In addition to computing a value, it computes how many times an expression of the form \texttt{count} \ e is evaluated. Here is the syntax and operational semantics:

\[ e ::= \lambda x. \ e \mid x \mid e \ e \mid \texttt{count} \ e \]

\[ c; e \rightarrow c'; e' \]

\[ \frac{c; \texttt{count} \ v \rightarrow c + 1; \ v}{c; \texttt{count} \ e \rightarrow c'; \texttt{count} \ e'} \]

Given a source program \( e \), our initial state is 0; \( e \) (i.e., the global count starts at 0). A program state \( c; e \) type-checks if \( e \) type-checks (i.e., the count can be any number).

(a) (5 points) Give a typing rule for \texttt{count} \ e that is sound and not unnecessarily restrictive.

\[ \quad \Gamma \vdash \text{T-COUNT} \]

\[ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{count} \ e : \tau} \]

(b) (10 points) State an appropriate Preservation Lemma for this language. Prove just the case(s) directly involving \texttt{count} \ e expressions.

(c) (10 points) State an appropriate Progress Lemma for this language. Prove just the case(s) directly involving \texttt{count} \ e expressions.

(d) (5 points) Give an example program that terminates in our language and would terminate if we changed function application to be call-by-name but under call-by-name it would produce a different resulting count\(^1\).

\begin{itemize}
  \item[(a)]
  \[ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{count} \ e : \tau} \]
\end{itemize}

\(^1\)Recall that in the call-by-value parameter passing mechanism the expression argument to a function is evaluated before the function is applied, while in call-by-name, the expression argument to a function is substituted for all the occurrences of the formal parameter and the resulting expression is then evaluated normally.

\[ 16 \]
(b) If $\cdot \vdash e : \tau$ and $c; e \rightarrow c'; e'$, then $\cdot \vdash e' : \tau$. We can prove this by induction on the derivation of $\cdot \vdash e : \tau$. In the case we’re asked to prove, the bottom of the derivation looks like:

$$
\begin{array}{c}
\cdot \vdash e_0 : \tau \\
\cdot \vdash \text{count } e_0 : \tau
\end{array}
$$

There are two possible ways $c; \text{count } e_0$ can step to some $e'$. If $e_0$ is a value, then $e' = e_0$ and the assumed derivation’s hypothesis $\cdot \vdash e_0 : \tau$ suffices. If $e_0$ is not a value, then $e' = \text{count } e'_0$ where $c; e_0 \rightarrow c'; e'_0$. So using $\cdot \vdash e_0 : \tau$ and induction, $\cdot \vdash e'_0 : \tau$, so we can derive $\cdot \vdash \text{count } e'_0 : \tau$.

(c) If $\cdot \vdash e : \tau$, then $e$ is a value or there exists an $e'$ and $c'$ such that $c; e \rightarrow c; e'$. In the case we’re asked to prove the bottom of the derivation looks like:

$$
\begin{array}{c}
\cdot \vdash e_0 : \tau \\
\cdot \vdash \text{count } e_0 : \tau
\end{array}
$$

So using $\cdot \vdash e_0 : \tau$, by induction either $e_0$ is a value or $c; e_0 \rightarrow c'; e'_0$ for some $c'$ and $e'_0$. If $e_0$ is a value, then $c; \text{count } e_0 \rightarrow c + 1; e_0$. If $c; e_0 \rightarrow c'; e'_0$, then we can derive $c; \text{count } e_0 \rightarrow c'; \text{count } e'_0$.

(d) One of an infinite number of examples is $(\lambda x. 0)(\text{count } 0)$. 