Final Exam
CSCI 5535, Fall 2016

December 3, 2016

Tasks (100 Points)

This exam consists of 100 points divided between OCaml-related tasks and proof-related tasks below. The OCaml tasks carry 25 points in total, whereas the proof tasks carry 75 points in total.

Below these two sets of tasks, there are additional tasks that you have the option of attempting to earn bonus points. These bonus tasks are optional; you need not attempt them to earn full credit on the exam.

OCaml Tasks (25 Points, total)

5 Points:
Define OCaml datatypes `card` and `cards` for the following inductive definitions:

$$\begin{array}{lrcl} \text{Card} & c & ::= & \text{Heart} \mid \text{Diamond} \mid \text{Spade} \mid \text{Club} \\ \text{Cards} & s & ::= & c :: s \mid \text{Empty} \end{array}$$

10 Points:
Define an OCaml function `unshuffle` with the following type:

```
unshuffle : ('a -> card -> (bool * 'a))
            -> 'a -> cards -> (cards * cards)
```

In particular, suppose a user runs your function as follows:

```
unshuffle choice init_state cards
```

Where the user chooses expressions for `choice`, `init_state` and `cards`.

Requirement 1: When the choice function `choice` returns `true`, the algorithm for `unshuffle` places the given card into the left deck of cards, and when it returns `false`, it places the card into the right deck.

Requirement 2: Further, `unshuffle` should use the choice state (of abstract type `'a`) that results from each choice to inform the next choice in the sequence of cards. The initial choice state `init_state` is always given by the user.

10 Points:
Define an OCaml function `fivefive` with the following type:

```
fivefive : cards -> (cards * cards)
```

This function should call `unshuffle` with a choice function and initial choice state that "unshuffles" the cards from the input list into two lists.

Requirement: The first five cards from the input deck should be placed in the left output deck. The next five cards should be placed into the right output deck. Each following ten cards should be distributed in the same way (five to the left, then five to the right). The number of cards may be zero or more, and need not be a multiple of ten or five. Until the cards end, `fivefive` follows the pattern described above.

Proof Tasks (75 Points, total)

Let us define a programming language **ExamLang**, with the following syntax, statics and dynamics (We covered this definition on Thursday Dec 1, 2016):

$$\begin{array}{rrcl} \text{Expressions} & e & ::= & \lambda x.e \mid e_1\ e_2 \mid e_1 + e_2 \mid n \mid x \\ \text{Types} & \tau & ::= & \texttt{num} \mid \tau_1 \to \tau_2 \end{array}$$

$\boxed{\Gamma \vdash e : \tau}$ "Under Γ, expression e has type τ"

$$\frac{(x:\tau) \in \Gamma}{\Gamma \vdash x : \tau}\ \text{S-VAR} \qquad \frac{}{\Gamma \vdash n : \texttt{num}}\ \text{S-NUM} \qquad \frac{\Gamma \vdash e_1 : \texttt{num} \qquad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash e_1 + e_2 : \texttt{num}}\ \text{S-PLUS}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \text{S-APP} \qquad \frac{\Gamma,\ x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}\ \text{S-LAM}$$

$\boxed{e\ \mathsf{val}}$ "Expression e is a value"

$$\frac{}{n\ \ \mathsf{val}}\ \text{V-NUM} \qquad \frac{}{\lambda x.e\ \ \mathsf{val}}\ \text{V-LAM}$$

$\boxed{e \longrightarrow e'}$ "expression e steps to expression e'"

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}\ \text{D-App1} \qquad \frac{e_2 \longrightarrow e_2'}{e_1\ e_2 \longrightarrow e_1\ e_2'}\ \text{D-App2} \qquad \frac{}{(\lambda x.e_1)\ e_2 \longrightarrow [e_2/x]e_1}\ \text{D-App3}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1 + e_2 \longrightarrow e_1' + e_2}\ \text{D-Plus1} \qquad \frac{e_2 \longrightarrow e_2'}{e_1 + e_2 \longrightarrow e_1 + e_2'}\ \text{D-Plus2} \qquad \frac{n_1 + n_2 = n_3}{n_1 + n_2 \longrightarrow n_3}\ \text{D-Plus3}$$

Proof Task 1 (15 Points):

- State (but do not prove) the substitution lemma for **ExamLang**
- State (but do not prove) the canonical forms lemma for **ExamLang**
- State the progress theorem for **ExamLang**. State over what structure to perform induction (but do not prove the cases).
- State the preservation theorem for **ExamLang**. State over what structure to perform induction (but do not prove the cases).
- Consider the proofs for progress and preservation for **ExamLang**. We discussed these in class on Dec 1. Answer these two questions:
  - In which theorem, and in which proof case do we need to apply the substitution lemma?
  - In which theorem, and in which proof case do we need to apply the canonical forms lemma?

Proof Task 2 (25 Points): Suppose that we extend **ExamLang** with let-bound variables, extending its abstract syntax with a new such form:

$$\text{Expressions} \quad e \quad ::= \quad \cdots \mid \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2$$

Subtasks: Your task is to update the definition and theorems for **ExamLang** to account for this new language construct for `let`.

- **Statics**: Extend the definition of $\Gamma \vdash e : \tau$ with one or more new rules.
- **Dynamics**: Extend the definition of $e \longrightarrow e_2$ with one or more new rules.
- **Canonical forms**: Does the canonical forms lemma (or its proof) need to change? If so, give the changes. If not, explain why not.
- **Substitution**: Does the substitution lemma statement need to change? If so, give the changes. If not, explain why not. (Do not prove any new cases).
- **Progress**: Give the *new* case(s) of the proof.
- **Preservation**: Give the *new* case(s) of the proof.

Proof Task 3 (35 Points): Suppose that we further extend our language with lazy computations, also known as thunks. We extend the syntax with new forms, as follows:

$$\begin{array}{rrcl} \text{Expressions} & e & ::= & \cdots \mid \mathsf{thunk}\ e \mid \mathsf{force}\ e \\ \text{Types} & \tau & ::= & \cdots \mid \mathsf{thunk}(\tau) \end{array}$$

For some intuition, recall that in homework #4, you use "stream thunks" to represent the "rest of the stream" in the tail position of each `Cons` cell. Recall that in that OCaml code, we represent these "stream thunks" as functions of type `unit -> stream`. These functions accept the unit value as an argument and they compute a stream value. To "force" these thunks to compute, we need only apply them to the unit value `()` of unit type `unit`.

More generally, thunks are values that represent suspended computations. When forced, thunks evaluate their suspended computation and produce another value.

To make this informal intuition of thunks precise *without encoding them as functions*, we extend the typing relation (statics), value relation and stepping relation (dynamics) with the following additional rules:

$\boxed{\Gamma \vdash e : \tau}$ "Under Γ, expression e has type τ"

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{thunk}\ e : \mathsf{thunk}(\tau)}\ \text{S-THUNK} \qquad \frac{\Gamma \vdash e : \mathsf{thunk}(\tau)}{\Gamma \vdash \mathsf{force}\ e : \tau}\ \text{S-FORCE}$$

$\boxed{e\ \mathsf{val}}$ "Expression e is a value"

$$\frac{}{\mathsf{thunk}\ e\ \ \mathsf{val}}\ \text{V-THUNK}$$

$\boxed{e \longrightarrow e'}$ "expression e steps to expression e'"

$$\frac{e \longrightarrow e'}{\mathsf{force}\ e \longrightarrow \mathsf{force}\ e'}\ \text{D-FORCE1} \qquad \frac{}{\mathsf{force}(\mathsf{thunk}\ e) \longrightarrow e}\ \text{D-FORCE2}$$

Subtasks: Your task is to update our lemmas, theorems and proofs, by doing all of the following:

- **Canonical forms**: Does the canonical forms lemma (or its proof) need to change? If so, give the changes. If not, explain why not.
- **Substitution**: Does the substitution lemma statement *or proof* need to change? If so, give the changes. (*Please give proofs for any new cases*). If not, explain why not.
- **Progress**: Give the *new* case(s) of the proof.
- **Preservation**: Give the *new* case(s) of the proof.

Bonus Tasks (35 Points):

Bonus Task 1 (15 Points): Characterize the asymptotic complexity of each of the following list functions, as discussed in class on October 27 (see `ocaml0.ml` on the lecture schedule).

For each, assume the input list(s) are of length $O(n)$. Your characterization should give the asymptotic time complexity, using big-O notation, for the number of reduction steps required to evaluate each of the following functions on such input lists:

- `split`
- `filter`
- `sum`
- `append` (assume both lists are of length $O(n)$)
- `reverse` (the version that uses `append`, internally)
- `reverse'` (the version that *does not* use `append`)
- `BubbleSort.bubble` (the standard bubble-sort algorithm using lists)

Bonus Task 2 (20 Points): Characterize the asymptotic complexity of each of the following stream functions, as described in Homework 4 (see `hw04.ml`), in the `StreamNil` module (defining *finite* streams).

For each function, assume the input streams(s) are finite, and of length $O(n)$. Your characterization should give the asymptotic time complexity, using big-O notation, for the number of reduction steps required evaluate to each of the following functions on such input lists.

Important hint: Unike lists, streams are lazy, since their tails consist of stream *thunks*. You should assume that creating a thunk requires only $O(1)$ time, regardless of the thunk's suspended computation.

- `singletons`
- `merge`
- `merge_adjacent`
- `sort_rec`
- `sort`