15–312: Principles of Programming Languages

Midterm Examination
(Sample Solutions)

October 18, 2018

- There are 7 pages in this examination, comprising 3 questions worth a total of 70 points.
- You may refer to your personal notes, course notes and supplements, and to Practical Foundations for Programming Languages, but not to any other person or source.
- You may use your laptop or tablet as long as you only refer to the aforementioned sources and disable WiFi and other network connections at all times.
- You have 80 minutes to complete this examination.
- Please answer all questions in the space provided with the question.
- There are two scratch sheets at the end for your use.

<table>
<thead>
<tr>
<th>Question:</th>
<th>Short</th>
<th>Safety</th>
<th>Dispatch</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points:</td>
<td>25</td>
<td>25</td>
<td>20</td>
<td>70</td>
</tr>
<tr>
<td>Score:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Question 1 [25]: Short Answers

(a) Choose the correct answer in each case. You may assume that basic arithmetic operations (+, ×, etc.) are defined.

i. [2 points] In \( T \), the value of

\[
\lambda(x : \text{nat}) \text{rec } x \{ z \mapsto \overline{1} | s(y) \mapsto x \times z \}
\]

is √ Itself. ○ Undefined. ○ Diverges. ○ The factorial of \( x \).

ii. [2 points] In \( \text{PCF} \) by-name, the value of

\[
\text{fix } f : \text{nat} \rightarrow \text{nat is } \lambda(x : \text{nat}) \text{ if } z \{ z \mapsto \overline{1} | s(x') \mapsto x \times f(x') \}
\]


(b) Following Barendregt, lists of natural numbers may be defined in the untyped \( \lambda \)-calculus as follows, making use of the definitions of the booleans and pairing given in homework 0.

- **nil** \( \triangleq \lambda x. x \).
- **cons** \( \triangleq \lambda h. \lambda t. \langle \text{F}, \langle h, t \rangle \rangle \).

Define the following operations on this representation of lists:

i. [2 points] Nil check:

\[
\text{isnil } \triangleq \lambda x. x \text{T}
\]

ii. [2 points] Head of a non-nil list:

\[
\text{hd } \triangleq \lambda x. x \text{F \ T}
\]

iii. [2 points] Tail of a non-nil list:

\[
\text{tl } \triangleq \lambda x. x \text{F \ F}
\]
(c) The coinductive type of infinite streams of natural numbers, \( \text{strm} \), has these operations:

- If \( e : \text{strm} \), then \( \text{unfold}(e) : \text{nat} \times \text{strm} \). Unfolding produces the next element and the rest of the stream.
- If \( \sigma \) is any state type, and \( e_0 : \sigma \) is the current state, and \( x : \sigma \vdash e : \text{nat} \times \sigma \) computes the next element and next state from a given state, then \( \text{gen}\{\sigma\}(x.e; e_0) : \text{strm} \) is a stream generated by the transition function and from a given state.

Define the following streams and operations:

i. [3 points] Infinite stream of zeros. Hint: the state is trivial.
\[
\text{gen}\{\text{unit}\}(x.(z, ));())
\]

ii. [3 points] Infinite stream of even numbers, starting with zero. Hints: the state is a natural number. You may take as given elementary arithmetic operations.
\[
\text{gen}\{\text{nat}\}(x.(2 \times x, x + 1); 0)
\]

iii. [4 points] Given a function \( f : \text{nat} \rightarrow \text{nat} \) and a stream \( s : \text{strm} \), define the stream that results from applying \( f \) to each element of \( s \). Hint: the state is a stream.
\[
\lambda (f : \text{nat} \rightarrow \text{nat}) \lambda (s : \text{strm}) \text{gen}\{\text{strm}\}(x.(f(\text{unfold}(s) \cdot l), \text{unfold}(s) \cdot r); s)
\]

(d) [5 points] Inductive types \( \mu(t.\tau) \) are introduced by \( \text{fold}\{t.\tau\}(e) \) and eliminated by the recursor \( \text{rec}\{\rho\}(x.e'; e) \). Define \( \text{unfold}(e) \) with type \( [\mu(t.\tau)/t]\tau \). Hint: the outermost form will be an instance of \( \text{rec} \); the inductive step will make use of \( \text{map}\{t.\tau\} \).

**Solution:** Define \( \text{unfold}(e) \) to be
\[
\text{rec}\{[\mu(t.\tau)/t]\tau\}(x.\text{map}\{t.\tau\}(y.\text{fold}\{t.\tau\}(y); x); e).
\]
The map re-folds the recursive calls in place to obtain the desired unfolding.
Question 2 [25]: Type Safety for Recursive Types

Recall these rules from the statics of FPC:

\[
\begin{align*}
\Delta, t \text{ type } & \vdash \tau \text{ type } \\
\Delta & \vdash \text{rec}(t.\tau) \text{ type } \\
\Gamma & \vdash \Delta \vdash \text{fold}\{t.\tau\}(e) : \text{rec}(t.\tau) \\
\Gamma & \vdash \Delta \vdash \text{unfold}(e) : \text{rec}(t.\tau)/t\tau
\end{align*}
\]

These are the corresponding rules for the eager dynamics of FPC:

\[
\begin{align*}
e & \text{val} \\
\text{fold}\{t.\tau\}(e) & \text{val} \\
e & \mapsto e' \\
\text{fold}\{t.\tau\}(e) & \mapsto \text{fold}\{t.\tau\}(e') \\
\text{unfold}(e) & \mapsto \text{unfold}(e') \\
\text{fold}(\text{fold}\{t.\tau\}(e)) & \mapsto e
\end{align*}
\]

Complete the following proofs for several cases of the preservation and progress theorems for FPC under the eager dynamics. The preservation, progress, and canonical forms theorems are stated as usual. You are to prove these cases of each of these:

(a) [5 points] Canonical Forms: If \( e : \text{rec}(t.\tau) \) and \( e \text{ val} \), then
\[
e = \text{fold}(e') \text{ and } e' \text{ val}
\]

(b) Preservation: \( \text{fold}\{t.\tau\}(e) \mapsto \text{fold}\{t.\tau\}(e') \) because \( e \mapsto e' \). Assume that \( \text{fold}\{t.\tau\}(e) : \sigma \); show that \( \text{fold}\{t.\tau\}(e') : \sigma \):

i. [2 points] By inversion of typing, \( \sigma = \text{rec}(t.\tau) \) and \( e : [\text{rec}(t.\tau)/t]\tau \)

ii. [2 points] Therefore by induction \( e' : [\text{rec}(t.\tau)/t]\tau \)

iii. [2 points] And so, by the statics, \( \text{fold}\{t.\tau\}(e') : \text{rec}(t.\tau) \)

(c) Preservation: \( \text{unfold}(\text{fold}\{t.\tau\}(e)) \mapsto e \).

i. [2 points] Assume that \( \text{unfold}(\text{fold}\{t.\tau\}(e)) : \sigma \)

ii. [2 points] Therefore, by inversion, \( \sigma = [\text{rec}(t.\tau)/t]\tau \)

and \( \text{fold}\{t.\tau\}(e) : \text{rec}(t.\tau) \)

iii. [2 points] Therefore, by inversion again, \( e : [\text{rec}(t.\tau)/t]\tau \)

(d) Progress: If \( \text{unfold}(e) : [\text{rec}(t.\tau)/t]\tau \) because \( e : \text{rec}(t.\tau) \).

i. [2 points] By induction either \( e \text{ val} \) or \( e \mapsto e' \)

ii. [2 points] In the former case, by canonical forms \( e = \text{fold}\{t.\tau\}(e') \) with \( e' \text{ val} \)

iii. [2 points] Therefore \( \text{unfold}(e) \mapsto e' \), as required.
iv. [2 points] In the latter case, it follows that $\text{fold}(e) \mapsto \text{fold}(e')$, which completes the proof.
Question 3 [20]: Dynamic Dispatch with Self-Referential Methods

The simple model of dynamic dispatch considered in class stresses the duality between sum and product types. Here we consider the extension of dynamic dispatch to account for recursive methods, which may call themselves via a self-referential variable standing for the entire collection of methods. As in the text we will consider two implementations of dynamic dispatch, one in which objects are tuples of methods, and one in which objects are tagged instance data.

This extension will be considered within an eager formulation of the language FPC of sums, products, partial functions, and recursive types. Rather than use self types, we will instead make explicit use of recursive types to implement self-reference. In this formulation the sender is responsible to apply the method to the object itself, as is often the case in implementations.

Let ρ be the recursive type \( \text{rec} \{ t \rightarrow \rho \}_{d \in D} \) representing a tuple of methods each of which is abstracted on a value of the type ρ itself. You are to assume that the type of the dispatch matrix is altered so that the entry for class c and method d has type \( \rho \rightarrow (\tau^c \rightarrow \rho) \).

The first argument is intended to be the method suite itself. Do not concern yourself with the dispatch matrix beyond the change of type just mentioned.

(a) In this question the representation type of objects is ρ, the recursive type given above.

You are to define the C-indexed tuple (that is, a tuple written in the form \( \langle e_c \rangle_{c \in C} \)) of new operations, \( e^1_{\text{new}} \), and the D-indexed tuple of snd operations, \( e^1_{\text{snd}} \), for this representation of objects. Your definition should reference the dispatch matrix \( DM \).

i. [7 points] Define \( e^1_{\text{new}} : (\tau^c \rightarrow \rho)_{c \in C} \). The methods within the resulting object are to be abstracted on an argument “this” of type ρ, which is to be provided by the sender at message send time.

Solution:

\[
e^1_{\text{new}} \triangleq \langle \lambda (y : \tau^c) \cdot \text{fold} (\langle d \rightarrow \lambda (\text{this} : \rho) \cdot (DM \cdot c \cdot d)(\text{this})(y) \rangle_{d \in D}) \rangle_{c \in C}
\]

ii. [3 points] Define \( e^1_{\text{snd}} : (\rho \rightarrow \rho)_{d \in D} \). The sender is responsible for supplying the object itself to the appropriate method.

Solution:

\[
e^1_{\text{snd}} \triangleq \langle \lambda (x : \rho) \cdot \text{unfold}(x) \cdot d(x) \rangle_{d \in D}
\]
(b) In this question the representation type of objects is the sum type $\sigma \triangleq [c \mapsto \tau^c]_{c \in C}$, as before. You are to define the $C$-indexed tuple of \texttt{new} operations, $e_{\text{new}}^\Pi$, and the $D$-indexed tuple of \texttt{snd} operations, $e_{\text{snd}}^\Pi$, for this representation of objects.

i. [2 points] Define $e_{\text{new}}^\Pi : \langle \tau^c \rightarrow \sigma \rangle_{c \in C}$. The result of calling \texttt{new} is to be a tagged data value.

Solution:

\[
e_{\text{new}}^\Pi \triangleq \langle c \mapsto \lambda (y : \tau^c) c \cdot y \rangle_{c \in C}
\]

ii. [8 points] Define $e_{\text{snd}}^\Pi : \langle \sigma \rightarrow \rho_d \rangle_{d \in D}$. The sender must supply the appropriate tuple of type $\rho$ to the chosen code in the dispatch matrix.

Solution:

\[
e_{\text{snd}}^\Pi \triangleq \langle \lambda (x : \sigma) \text{ unfold}(e) \cdot d(e) \rangle_{d \in D}, \text{ where } e \triangleq \text{ fold} \langle \lambda (this : \rho) \text{ case } x \{ c \cdot y \mapsto (DM \cdot c \cdot d)(this)(y) \} \rangle_{c \in C} \rangle_{d \in D}
\]