15–312: Principles of Programming Languages

Midterm Examination

March 3, 2016

- There are 14 pages in this examination, comprising 4 questions worth a total of 85 points.
- You may refer to your personal notes and to Practical Foundations of Programming Languages, but not to any other person or source.
- You have 80 minutes to complete this examination.
- Please answer all questions in the space provided with the question.
- There are three scratch sheets at the end for your use.

Full Name: ____________________________________________

Andrew ID: ____________________________________________

<table>
<thead>
<tr>
<th>Question</th>
<th>Short Answer</th>
<th>Type Safety</th>
<th>Inductive Types</th>
<th>Data Abstraction</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points:</td>
<td>20</td>
<td>20</td>
<td>30</td>
<td>15</td>
<td>85</td>
</tr>
<tr>
<td>Score:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Question 1 [20]: Short Answers

Consider the sort tm of terms generated by the following operators:

\[
\begin{align*}
\text{ap} & : (tm, tm) \to tm \\
\lambda & : (tm, tm) \to tm
\end{align*}
\]

These are called *applications* and *\(\lambda\)-abstractions*, respectively. Variables \(u, v, w, x, y, z\) are all implicitly of sort tm.

(a) Answer “true” or “false” for each of the following purported \(\alpha\)-equivalences:

i. \(1\) point \(\text{ap}(x, y) =_\alpha \text{ap}(y, x)\).

ii. \(1\) point \(\lambda(x.\lambda(y.x)) =_\alpha \lambda(u.\lambda(v.v))\).

iii. \(1\) point \(\lambda(x.\text{ap}(\lambda(y.y), \lambda(z.x))) =_\alpha \lambda(y.\text{ap}(\lambda(z.x), \lambda(z.y)))\).

iv. \(1\) point \(\text{ap}(\lambda(x.x), \lambda(x.x)) =_\alpha \lambda(x.x)\).

(b) Calculate the result of the following substitutions on terms identified up to \(\alpha\)-equivalence:

i. \(1\) point \([\lambda(x.x)/y]z\).

ii. \(1\) point \([\lambda(x.x)/y]y\).

iii. \(1\) point \([\lambda(x.x)/y]\lambda(x.y)\).

iv. \(1\) point \([\lambda(x.y)/z]\lambda(y.\text{ap}(y, z))\).
(c)  

i. **6 points** The relation $t \rightarrow_{\beta} t'$, called $\beta$-reduction, is inductively defined by a collection of rules that includes the axiom

$\text{ap}(\lambda(x.t), t') \rightarrow_{\beta} [t'/x]t$.

Give the remaining rules defining $t \rightarrow_{\beta} t'$ with the meaning that the above step may be anywhere within $t$, including either argument of an application, and with the body of a $\lambda$, to obtain $t'$.

ii. **6 points** Give a term $t$ such that $t \rightarrow_{\beta} t$; that is, $t$ $\beta$-reduces to itself. **Hint:** use self-application and the given axiom of $\beta$-reduction.
Question 2 [20]: Type Safety

Recursive types represent self-referential data structures using fold and unfold operations. For example, the type

\[ \text{list} \triangleq \text{rec is unit + (nat} \times t) \]

is, under an eager interpretation, the type of finite lists of natural numbers. A list is created using fold. For example, the one-element list containing the number 7 is written

\[ \text{fold}(r \cdot \langle 7, \text{fold}(1 \cdot \langle \rangle) \rangle) . \]

Lists are decomposed using unfold to expose their structure, and then a case analysis to determine its structure.

Languages that lack sums and recursive types use pointers to represent self-referential data structures. For example, in C we might write

```c
struct node {
    int hd;
    node *tl;
};
typedef struct node *list;
```

to represent the type of lists as pointers to nodes. The empty list is represented by the “null pointer”, NULL; a non-empty list is allocated by initializing a region of memory in the format of a node and returning a pointer to it. A list is decomposed by following the pointer and accessing the components of the structure.

An abstract form of such a representation may be given using pointer types and self-referential definitions. Thus, we may define

\[ \text{list} \triangleq *\text{(nat} \times \text{list}) \]

much in the manner of C. The length-one list containing the number 7 is written

\[ &\langle 7, \text{NULL} \rangle . \]

The components of a list \( x \) would be accessed by writing \(*x \cdot l\) and \(*x \cdot r\).

(Question continues on the next page.)
Consider the following statics and dynamics of pointer types:

$$\begin{align*}
\Gamma \vdash \text{NULL} : *\tau & \quad \Gamma \vdash e : \tau & \quad \Gamma \vdash e : *\tau \\
\Gamma \vdash \&e : *\tau & \quad \Gamma \vdash *e : \tau
\end{align*}$$

\[
\begin{array}{c|c|c|c|c|c|c}
\text{NULL val} & e \ 	ext{val} & \&e \ 	ext{val} & e \mapsto e' & \&e \mapsto \&e' & *e \mapsto *e' & *\&e \mapsto \, e
\end{array}
\]

(a) Prove the preservation property for pointer types: if $e : \tau$ and $e \mapsto e'$, then $e' : \tau$. Proceed by induction on the derivation of $e \mapsto e'$, considering only the rules of the dynamics given above.

i. **5 points** Case $\&e \mapsto \&e'$ because $e \mapsto e'$:

ii. **5 points** Case $*e \mapsto *e'$ because $e \mapsto e'$:

iii. **5 points** Case $*\&e \mapsto e$, where $e \ 	ext{val}$:
(b) [5 points] Prove or disprove the progress property for pointer types: if \( e : \tau \), then either \( e \text{ val} \) or \( e \rightarrow e' \) for some \( e' \). A proof should proceed by induction on the derivation of \( e : \tau \); a disproof should consist of a counterexample and a proof that it is such.
The natural numbers, as defined in $\mathbf{T}$, form the prime example of an inductive type. Recall that the natural numbers may be encoded in $\mathbf{F}$ by the type

$$\text{nat} \triangleq \forall t. t \to (t \to t) \to t.$$ 

Each number $n$ is represented by its own iterator that applies a given transformation $n$ times starting with a given basis.

The transfinite ordinals extend the finite ordinals beyond infinity by adding a new constructor, $\text{sup}(x.e)$, in addition to zero and successor, with the following typing rules:

$$\begin{align*}
\Gamma \vdash z : \text{ord} & \quad \Gamma \vdash e : \text{ord} & \quad \Gamma, x : \text{nat} \vdash e : \text{ord} \\
\Gamma \vdash \text{sup}(x.e) : \text{ord}
\end{align*}$$

Informally, the ordinal $\text{sup}(x.e)$ is the supremum (least upper bound) of the infinite sequence of ordinals $[0/x]e, [1/x]e, [2/x]e, \ldots$. Thinking of ordinals as trees, $z$ is a leaf, $s(e)$ is a node with one child, and $\text{sup}(x.e)$ is a node with an entire sequence of children. Whereas a successor has one predecessor, a supremum has infinitely many predecessors.

Each natural number $n : \text{nat}$ may be regarded as a finite ordinal $\hat{n} : \text{ord}$ defined in the obvious way (the natural number zero is the ordinal zero, and the successor of a natural number is the ordinal successor of that number regarded as an ordinal). Using this notation, the least transfinite ordinal, $\omega$, is defined to be $\text{sup}(x.\hat{x})$, the ordinal that comes immediately after all the finite ordinals.

Because it is built up from the three constructors given, an ordinal can be infinitely “wide”, but only finitely “deep.” This means that we may define an ordinal iterator that generalizes the natural numbers iterator in $\mathbf{T}$. It has three branches, one for $z$, one for $s(e)$, and one for $\text{sup}(x.e)$. The successor branch may use its own result on the predecessor, and the supremum branch may use its own result on any (perhaps many) of its predecessors.

The statics of the ordinal iterator is given by the following rule:

$$\begin{align*}
\Gamma \vdash e : \text{ord} & \quad \Gamma \vdash e_0 : \tau & \quad \Gamma, x : \tau \vdash e_1 : \tau & \quad \Gamma, x : \text{nat} \to \tau \vdash e_2 : \tau \\
\Gamma \vdash \text{ordrec}\{\tau\}{e_0}{x.e_1}{x.e_2}(e) : \tau
\end{align*}$$

The first two premises are the same as for the natural numbers iterator; the third expresses that the result for a supremum is defined in terms of the results for each of its countably many predecessors.
(a) 10 points The dynamics of the ordinal iterator consists of the following two rules for zero and successor, plus one more than you are to fill in:

\[
\text{ordrec}\{\tau\}\{e_0; x.e_1; x.e_2\}(z) \mapsto e_0
\]

\[
\text{ordrec}\{\tau\}\{e_0; x.e_1; x.e_2\}(s(e)) \mapsto \text{ordrec}\{\tau\}\{e_0; x.e_1; x.e_2\}(e)/x[e_1]
\]

State the rule for the dynamics of the ordinal iterator on the supremum. *Hint:* Form the infinite sequence of results of the recursive calls on each of the predecessors, as described by the statics.

(b) The transfinite ordinals may be encoded into $F$ as a polymorphic type by generalizing the encoding of natural numbers to account for the supremum. *Hint:* Follow the pattern used to define $\text{nat}$ in $F$.
  i. 5 points Define $\text{ord}$ as a polymorphic type:

  ii. 5 points Define the function $\text{hat} : \text{nat} \to \text{ord}$ that converts each natural number into the corresponding ordinal. *Hint:* Remember that the natural numbers are encoded in $F$.

(Question continues on the next page)
iii. 5 points Define the function $\text{sup} : (\text{nat} \to \text{ord}) \to \text{ord}$ that forms the supremum of a sequence of ordinals. Hint: Follow the same logic as used to define the successor operation on natural numbers encoded in $\text{F}$. 

iv. 5 points Using $\text{hat}$ and $\text{sup}$, define $\omega : \text{ord}$, the first transfinite ordinal, in $\text{F}$. 
Question 4 [15]: Data Abstraction

Dynamic dispatch is an example of data abstraction given by the following existential type:

$$\exists (t_{\text{obj}} \mapsto \prod_{c \in C} \tau^c \rightarrow t_{\text{obj}}, \text{snd} \mapsto \prod_{d \in D} t_{\text{obj}} \rightarrow \rho_d)).$$

That is, there is a type of objects admitting a \texttt{new} operation for each class \(c \in C\) and a \texttt{snd} operation for each method \(d \in D\). We described two different implementations of this interface:

\[
\begin{align*}
\tau_{\text{CB}} & \triangleq \prod_{d \in D} \rho_d & \tau_{\text{MB}} & \triangleq \sum_{c \in C} \tau^c \\
\text{new}[c](e^c) & \doteq (e_{\text{dm}} \cdot c \cdot d(e^c))_{d \in D} & \text{new}[c](e^c) & \doteq c \cdot e^c \\
\text{snd}[d](e) & \doteq e \cdot d & \text{snd}[d](e) & \doteq \text{case } c \{ c \cdot u \mapsto e_{\text{dm}} \cdot c \cdot d(u) \}_{c \in C}.
\end{align*}
\]

The class-based and the method-based implementations are equivalent. Specifically, they both assign the same behavior to each method on each class instance. To prove their equivalence, it suffices to define a \textit{bisimulation relation} between their representation types, as described in PFPL Chapter 17. In the present case this means that we must do three things:

1. Define a relation \(\mathcal{R}\) between values of type \(\tau_{\text{CB}}\) and \(\tau_{\text{MB}}\), the respective implementation types for the dynamic dispatch abstraction.

2. Show that for each \(c \in C\), the result \(\text{new}[c](e^c)\) in the class-based implementation is related to the result of \(\text{new}[c](e^c)\) in the method-based implementation are related by \(\mathcal{R}\).

3. Show that for each \(d \in D\), if \(e_{\text{CB}} : \tau_{\text{CB}}\) and \(e_{\text{MB}} : \tau_{\text{MB}}\) are related by \(\mathcal{R}\), then the result of \(\text{snd}[e_{\text{CB}}](d)\) in the class-based implementation is equal to the result of \(\text{snd}[e_{\text{MB}}](d)\) in the method-based implementation.

(Question continues on the next page.)
(a) Define the bisimulation relation $R$ by completing the following definition:

\[ e_{cb} \mathcal{R} e_{mb} \text{ iff } \ldots \]

*Hint:* your solution will refer to the dispatch matrix and will make use of all of the information available on the left and right sides of the relation, including the representation types.

(b) Show that `new` creates $\mathcal{R}$-related objects from the same instance data.

(c) Show that `snd` yields equal results on $\mathcal{R}$-related objects.
Scratch Work:
Scratch Work:
Scratch Work: