15–312: Principles of Programming Languages

Midterm Examination
Sample Solutions

March 5, 2015

• There are 10 pages in this examination, comprising 4 questions worth a total of 130 points.

• You have 80 minutes to complete this examination.

• Please answer all questions in the space provided with the question.

• You may refer to on-paper notes and to the text, but to no other person or source, during the examination.

Full Name:  

<table>
<thead>
<tr>
<th>Question</th>
<th>Arithmetic</th>
<th>Union Types</th>
<th>Lambda Calculus</th>
<th>Dynamic Typing</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points:</td>
<td>20</td>
<td>65</td>
<td>25</td>
<td>20</td>
<td>130</td>
</tr>
<tr>
<td>Score:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Question 1 [20]: Arithmetic

(a) [10 points] The Stirling numbers are defined by the recurrence:

\[
\begin{align*}
S(0, 0) &= 1 \\
S(n, 0) &= 0 \quad \text{(for } n > 0) \\
S(0, k) &= 0 \quad \text{(for } k > 0) \\
S(n, k) &= k \cdot S(n-1, k) + S(n-1, k-1) \quad \text{(for } n, k > 0)
\end{align*}
\]

Give code computing the Stirling numbers as a function \( \text{nat} \to \text{nat} \to \text{nat} \) in PCF. Take addition and multiplication as given.

Solution:

\[
\text{fix } \text{Stirling} : \text{nat} \to \text{nat} \to \text{nat}.
\lambda n : \text{nat}.
\lambda k : \text{nat}.
\begin{align*}
\text{ifz } n \{ z \Rightarrow & \text{ifz } k \{ z \Rightarrow s(z) \mid s(\_ ) \Rightarrow z \} \mid \\
\text{s}(n') \Rightarrow & \text{ifz } k \{ z \Rightarrow z \mid s(k') \Rightarrow k \ast \text{Stirling } n' \ast k + \text{Stirling } n' \ast k' \} \}
\end{align*}
\]

(b) [10 points] Give code computing the Stirling numbers in Gödel’s T. Take addition and multiplication as given.

Solution: \( \lambda n : \text{nat}. \)

\[
\text{rec } n \{ z \Rightarrow \lambda k : \text{nat}. \text{rec } k \{ z \Rightarrow s(z) \mid s(\_ ) \Rightarrow z \} \mid \\
\text{s}(n') \Rightarrow \lambda k : \text{nat}. \text{rec } k \{ z \Rightarrow z \mid s(k') \Rightarrow k \ast f k + f k' \} \}
\]

The following function is not a solution:

\[
\lambda n : \text{nat}. \lambda k : \text{nat}.
\text{rec } n \{ z \Rightarrow \text{rec } k \{ z \Rightarrow s(z) \mid s(\_ ) \Rightarrow z \} \mid \\
\text{s}(n') \Rightarrow \text{rec } k \{ z \Rightarrow z \mid s(k') \Rightarrow k \ast x + y \} \}
\]

Let \( f \) be the above. Then \( f \overset{\bar{n} \bar{k}}{\rightarrow} * z \) for all \( k > 0 \).
**Question 2 [65]: Union Types**

The sum type $\tau_1 + \tau_2$ is also sometimes referred to as the “disjoint union” or “tagged union” of $\tau_1$ and $\tau_2$, since the values of a sum type $l \cdot e$ and $r \cdot e$ contain tags (i.e. the $l$ or $r$) indicating to which arm of the sum a value belongs.

A related, but very different, concept is the “union” type (or “untagged union” for emphasis), which is typically written $\tau_1 \lor \tau_2$. A value of $\tau_1 \lor \tau_2$ is simply a value of $\tau_1$ or of $\tau_2$, without any tag indicating which of the two it is.

An untagged union can be made part of a sound type system using sophisticated type theory. In this problem, we will examine an unsophisticated version, and consider its ramifications.

Let the syntax of $L_\lor$ be as follows:

```
(types)          \tau ::= \cdots \mid \tau \lor \tau
(expressions)   \ e ::= \cdots \mid \text{in } e \mid \text{ucase}(e,x.e,x.e)
```

Note that the introduction form in contains no tag indicating left or right. The static semantics are as follows:

\[
\Gamma \vdash e : \tau_1 \\
\Gamma \vdash \text{in } e : \tau_1 \lor \tau_2 \tag{1}
\]
\[
\Gamma \vdash e : \tau_2 \\
\Gamma \vdash \text{in } e : \tau_1 \lor \tau_2 \tag{2}
\]
\[
\Gamma \vdash e : \tau_1 \lor \tau_2 \quad \Gamma, x : \tau_1 \vdash e_1 : \tau \\
\Gamma, x : \tau_2 \vdash e_2 : \tau \\
\Gamma \vdash \text{ucase}(e,x.e_1,x.e_2) : \tau \tag{3}
\]

The operational semantics are as follows:

\[
\text{in } e \text{ val} \quad \text{ucase}(e,x.e_1,x.e_2) \text{ val} \quad \text{ucase}(e,x.e_1,x.e_2) \hookrightarrow \text{ucase}(e',x.e_1,x.e_2) \tag{4} \tag{5} \tag{6}
\]
\[
\text{ucase}(\text{in } e,x.e_1,x.e_2) \hookrightarrow [e/x]e_1 \tag{7} \tag{8}
\]

This language, $L_\lor$, is not type-safe. Consequently, either preservation or progress or both must fail to hold. As it turns out, one of them is true and the other is false.
(a) 25 points Indicate which of preservation or progress is false for $\mathcal{L}_\vee$. Give a counterexample. (You may assume the presence of some other types we have studied for the purpose of your counterexample.)

**Solution:** Preservation is false. Consider the expression $\text{ucase}(\text{in} \bar{0}, x.x, f.f(\bar{12}))$. It has the type $\text{nat}$:

\[
\begin{array}{c}
\bar{0} : \text{nat} \\
\text{in} \bar{0} : \text{nat} \lor (\text{nat} \rightarrow \text{nat}) \\
x : \text{nat} \vdash x : \text{nat} \\
f : (\text{nat} \rightarrow \text{nat}) \vdash f(\bar{12}) : \text{nat} \\
\text{ucase}(\text{in} \bar{0}, x.x, f.f(\bar{12})) : \text{nat}
\end{array}
\]

However $\text{ucase}(\text{in} \bar{0}, x.x, f.f(\bar{12})) \mapsto [\bar{0}/f](f(\bar{12})) = \bar{0}(\bar{12})$, which is ill-typed.

(b) 30 points Prove either preservation or progress for $\mathcal{L}_\vee$. (Choose carefully which one you attempt to prove. One of them is impossible!) You may assume a canonical forms lemma or an inversion lemma, as appropriate, but be clear what your lemma says. Please give only the case for rule 3 or 7 (as appropriate).

**Solution:** The proof is by induction on the derivation of $\Gamma \vdash e : \tau$.

**Case:** When the last rule applied is:

\[
\begin{array}{c}
\Gamma \vdash e_0 : \tau_1 \lor \tau_2 \\
\Gamma, x : \tau_1 \vdash e_1 : \tau \\
\Gamma, x : \tau_2 \vdash e_2 : \tau
\end{array}
\]

\[\Gamma \vdash \text{ucase}(e_0, x.e_1, x.e_2) : \tau\] (3)

We wish to show that $\text{ucase}(e_0, x.e_1, x.e_2) \text{ val}$, or $\text{ucase}(e_0, x.e_1, x.e_2) \mapsto e'$ for some $e'$. By assumption, $\Gamma = \cdot$. By the induction hypothesis, either $e_0 \text{ val}$ or $e_0 \mapsto e'_0$.

Suppose $e_0 \mapsto e'_0$. Then $\text{ucase}(e_0, x.e_1, x.e_2) \mapsto \text{ucase}(e'_0, x.e_1, x.e_2)$ using rule (6).

Alternatively, suppose $e_0 \text{ val}$. Since $e_0 \text{ val}$ and $\cdot \vdash e_0 : \tau_1 \lor \tau_2$, by Canonical Forms we obtain that $e_0$ has the form $\text{in} e'_0$. Then $\text{ucase}(e_0, x.e_1, x.e_2) \mapsto [e'_0/x]e_1$ using rule (7).

(c) 10 points Two other properties we discussed are determinism and unique typing. One or both of them fails to hold for $\mathcal{L}_\vee$. Give a counterexample for one of them.

**Solution:** Determinism fails: Observe that $\text{ucase}(\text{in} \bar{0}, x.x, x.s(x))$ steps to both $\bar{0}$ and $\bar{1}$.

Unique typing fails: Observe that $\text{in} \bar{0}$ has the type $\text{nat} \lor \tau$ (and also $\tau \lor \text{nat}$) for any type $\tau$. 

15–312 Midterm 4 of 10 © 2015 Carnegie Mellon University
Question 3 [25]: Lambda Calculus

This question concerns the untyped \( \lambda \)-calculus. Recall that the Church numerals are a representation of the natural numbers as untyped \( \lambda \)-terms. Specifically, the number \( n \in \mathbb{N} \) is represented by these equations:

\[
0 \triangleq \lambda (z) \lambda (s) z \\
\overline{n + 1} \triangleq \lambda (z) \lambda (s) s(\overline{n}(z)(s)).
\]

The successor function, \( \text{succ} \), may be defined on Church numerals by the equation

\[
\text{succ} \triangleq \lambda (x) \lambda (z) \lambda (s) s(x(z))(s)
\]

so that the second equation may be written

\[
\overline{n + 1} \triangleq \text{succ}(\overline{n}).
\]

The predecessor function, \( \text{pred} \), is also definable, as we saw in class.

Solve both of the following problems without using the \( \text{Y} \) combinator or any form of self-application. You may use \( \text{pred} \) and \( \text{succ} \).

(a) 10 points Define the \( \lambda \)-term \( \text{max} \) such that \( \text{max}(\overline{m})(\overline{n}) \equiv p \), where \( p \) is the larger of \( m \) and \( n \), using the following identity for the maximum function on natural numbers:

\[
\max(x, y) = (x - y) + y
\]

(This works because on natural numbers \( x - y = 0 \) whenever \( x \leq y \).)

Solution:

\[
\text{max} \triangleq \lambda (x) \lambda (y) (y x \text{pred}) \text{succ}
\]

(b) 5 points A Church-like representation \( \overline{t} \) of bare binary trees \( t \) may be given as follows:

- The empty tree \( \varepsilon \) is given by the equation

\[
\varepsilon \triangleq \lambda (e) \lambda (n) e,
\]

which simply returns the result for an empty tree.

- The non-empty tree \( t = \text{node}(t_1, t_2) \) with children \( t_1 \) and \( t_2 \) is given by the equation

\[
\overline{t} \triangleq \lambda (e) \lambda (n) n(\overline{t_1}(e)(n))(\overline{t_2}(e)(n)),
\]

which recursively applies the children to the given basis, \( e \), and inductive step, \( n \), and calls \( n \) on their results.

Give the Church representation of the binary tree consisting of one node with two empty children. (Your solution should be an explicit \( \lambda \)-term in simplest form.)
(c) [10 points] Define the $\lambda$-term $\text{height}$ such that $\text{height}(t) \equiv \overline{n}$, where $n$ is the height of the binary tree $t$. *Hint:* Use the function $\text{max}$ defined in the previous part of this question.

**Solution:** Define $\text{height}$ by the following equation:

$$\text{height} \triangleq \lambda(t) \cdot t(\overline{0})\left(\lambda(h_1) \cdot \lambda(h_2) \cdot \text{succ}(\text{max}(h_1)(h_2))\right).$$

The height of the empty tree is zero. The height of a non-empty tree with two children is one more than the maximum of the heights of the two children.
Question 4 [20]: Dynamic Typing

In this question you are to consider various aspects of dynamic typing. In homework you considered the following extension to PCF with these Lisp-like primitives for the null tuple and pairing:

<table>
<thead>
<tr>
<th>Sort</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exp d ::=</td>
<td>nil</td>
<td>nil</td>
</tr>
<tr>
<td></td>
<td>cons(d₁;d₂)</td>
<td>cons(d₁;d₂)</td>
</tr>
<tr>
<td></td>
<td>car(d)</td>
<td>car(d)</td>
</tr>
<tr>
<td></td>
<td>cdr(d)</td>
<td>cdr(d)</td>
</tr>
<tr>
<td></td>
<td>cond(d;d₀;d₁)</td>
<td>cond(d;d₀;d₁)</td>
</tr>
</tbody>
</table>

Using these primitives one may represent lists by iterated pairing, ending with the null tuple, so that the list \([d₁, \ldots, dₙ]\) is represented by the expression

\[
\text{cons}(d₁; \ldots \text{cons}(dₙ;\text{nil})).
\]

If \(d\) is such a list, then \(\text{car}(d)\) is its head, and \(\text{cdr}(d)\) is its tail. The conditional \(\text{cond}(d;d₀;d₁)\) evaluates to the value of \(d₁\) if \(d\) evaluates to \(\text{nil}\), and to the value of \(d₀\) otherwise.

For your reference, the dynamics of this extension is given by the following rules, with the rules of error propagation omitted, and disregarding the interaction with the other aspects of PCF.

\[
\begin{align*}
\text{nil is nil} \quad & \quad (1a) \\
\text{nil is not cons} \quad & \quad (1b) \\
\text{nil is val} \quad & \quad (1c) \\
\text{d₁ is val, d₂ is val} \quad \text{cons}(d₁;d₂) \text{ is cons } d₁, d₂ \quad & \quad (1d) \\
\text{d₁ is val, d₂ is val} \quad \text{cons}(d₁;d₂) \text{ is not nil} \quad & \quad (1e) \\
\text{d₁ is val, d₂ is val} \quad \text{cons}(d₁;d₂) \text{ is val} \quad & \quad (1f) \\
\text{d \mapsto d'} \quad \text{car}(d) \mapsto \text{car}(d') \quad & \quad (1g) \\
\text{d is cons d₁, d₂} \quad \text{car}(d) \mapsto d₁ \quad & \quad (1h) \\
\text{d is val, d is not cons} \quad \text{car}(d) \mapsto \text{err} \quad & \quad (1i)
\end{align*}
\]
(a) 10 points Usually \texttt{cons}(d_1; d_2) imposes no restrictions on either of its arguments so that a pair of any two values may be formed using it. One consequence is that one may form a value such as \texttt{cons}(0; \texttt{cons}(1; 2)), which is not a list because it does not end with \texttt{nil}. Suppose instead that one wished to enforce the invariant that one may form \texttt{cons}(d_1; d_2) only if \(d_2\) is either \texttt{nil} or \texttt{cons} whose second component again satisfies this requirement. Give a dynamics that enforces this invariant. Consider only those rules pertaining to \texttt{nil} and \texttt{cons}. You may omit any error propagation rules, but must consider any error generation rules.

Solution:

\[
\frac{d \mapsto d'}{\texttt{cdr}(d) \mapsto \texttt{cdr}(d')}
\]

\[
\frac{d \text{ is\_cons } d_1, d_2}{\texttt{cdr}(d) \mapsto d_2}
\]

\[
\frac{d \text{ val } d \text{ is\_not\_cons}}{\texttt{cdr}(d) \text{ err}}
\]

\[
\frac{d \mapsto d'}{\texttt{cond}(d; d_0; d_1) \mapsto \texttt{cond}(d'; d_0; d_1)}
\]

\[
\frac{d \text{ val } d \text{ is\_not\_nil}}{\texttt{cond}(d; d_0; d_1) \mapsto d_0}
\]

\[
\frac{d \text{ val } d \text{ is\_nil}}{\texttt{cond}(d; d_0; d_1) \mapsto d_1}
\]

\[
\frac{d_1 \mapsto d_1'}{\texttt{cons}(d_1; d_2) \mapsto \texttt{cons}(d_1'; d_2)}
\]

\[
\frac{d_1 \text{ val } d_2 \mapsto d_2'}{\texttt{cons}(d_1; d_2) \mapsto \texttt{cons}(d_1'; d_2)}
\]

\[
\frac{d_1 \text{ val } d_2 \text{ val } d_2 \text{ is\_nil}}{\texttt{cons}(d_1; d_2) \text{ val}}
\]

\[
\frac{d_1 \text{ val } d_2 \text{ val } d_2 \text{ is\_cons \_\_\_}}{\texttt{cons}(d_1; d_2) \text{ val}}
\]

\[
\frac{d_1 \text{ val } d_2 \text{ val } d_2 \text{ is\_not\_nil } d_2 \text{ is\_not\_cons}}{\texttt{cons}(d_1; d_2) \text{ err}}
\]

(b) Using the Lisp-like primitives introduced above, the append function, \texttt{app}, may be defined by the following equation:

\[
\texttt{app} \triangleq \text{\texttt{fix a is } \lambda(x) \lambda(y) \texttt{cond}(x; \texttt{cons(car}(x); a(\texttt{cdr}(x))(y)); y).}
\]

Which class checks in the above code are strictly unnecessary under the following two separate conditions:
i. 5 points The liberal interpretation of \texttt{cons}, which imposes no checks on its arguments.

\textbf{Solution:} The check that \texttt{a} is a function at the application site is strictly redundant, because the use of self-reference ensures that it will always come true. The check on the argument to \texttt{car} is required, because the conditional only ensures that it is not \texttt{nil}, but not that it is a \texttt{cons}. The check on the argument to \texttt{cdr} is strictly redundant, because the earlier check on \texttt{car} guarantees \texttt{x} is a \texttt{cons}.

ii. 5 points The strict interpretation of \texttt{cons}, which enforces the invariant that only proper lists may be formed using it.

\textbf{Solution:} The check imposed by \texttt{cons} is not redundant because \(a(cdr(x))(y)\) can return \texttt{y}, which is unconstrained.
Reference

Gödel’s T

\[ \Gamma, x : \tau \vdash x : \tau \]

\[ \Gamma \vdash z : \text{nat} \quad \Gamma \vdash e : \text{nat} \quad \Gamma \vdash s(e) : \text{nat} \]

\[ \Gamma \vdash e : \text{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \text{nat}, y : \tau \vdash e_1 : \tau \]

\[ \Gamma \vdash \text{rec}(e; e_0; x . y. e_1) : \tau \]

\[ \Gamma \vdash \lambda(x : \tau_1) e : \tau_1 \rightarrow \tau_2 \]

\[ \Gamma \vdash \lambda(x : \tau_1) e : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_2 : \tau \]

\[ \Gamma \vdash e_1 e_2 : \tau \]

PCF

\[ \Gamma, x : \tau \vdash x : \tau \]

\[ \Gamma \vdash z : \text{nat} \quad \Gamma \vdash e : \text{nat} \quad \Gamma \vdash s(e) : \text{nat} \]

\[ \Gamma \vdash e : \text{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \text{nat} \vdash e_1 : \tau \]

\[ \Gamma \vdash \text{ifz}(e; e_0; x . e_1) : \tau \]

\[ \Gamma \vdash \lambda(x : \tau_1) e : \tau_1 \rightarrow \tau_2 \]

\[ \Gamma \vdash \lambda(x : \tau_1) e : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_2 : \tau \]

\[ \Gamma \vdash e_1 e_2 : \tau \]

\[ \Gamma, x : \tau \vdash e : \tau \]

\[ \Gamma \vdash \text{fix} x : \tau . e : \tau \]