15–312: Principles of Programming Languages

Midterm Examination
Sample Solutions
March 6, 2014

• There are 11 pages in this examination, comprising 3 questions worth a total of 100 points.
• You have 80 minutes to complete this examination.
• Please answer all questions in the space provided with the question.
• There are scratch pages at the end for your use.
• You may refer to your personal notes and to the text, but to no other person or source, during the examination.

Full Name:  

Andrew ID:  

<table>
<thead>
<tr>
<th>Question</th>
<th>Short Answer</th>
<th>Streams</th>
<th>Stack computation</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points:</td>
<td>30</td>
<td>40</td>
<td>30</td>
<td>100</td>
</tr>
<tr>
<td>Score:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Question 1 [30]: Short Answer

(a) (6 points) Abstract Binding Trees
We can represent the (uni/un)typed λ-calculus as an abstract binding tree with two operators, \( \text{lam}(x.e) \) and \( \text{ap}(e_1; e_2) \), with the usual concrete syntax \( \lambda(x)e \) and \( e_1(e_2) \), respectively. Give the result of performing the following substitution. (Both concrete and abstract syntax are given.)

Concrete Syntax: \[ \frac{(\lambda(x)f(x))/y}{(\lambda(y)f(y))} \quad (\lambda(y)x(y)) \]
ABT Syntax: \[ \text{ap}(\text{lam}(x.\text{ap}(f;x))/y) \quad \text{ap}(\text{lam}(f.\text{ap}(f;y)); \text{lam}(y.\text{ap}(x;y))) \]

Only perform the substitution as defined on abstract binding trees, do not reduce according to the dynamics of the λ-calculus.

Solution:
Key idea: we have to alpha-vary the bound instance of \( f \) to something new:

Concrete Syntax: \[ (\lambda(g)(\lambda(x)f(x)))/y \quad (\lambda(y)x(y)) \]
ABT Syntax: \[ \text{ap}(\text{lam}(g.\text{ap}(g;\text{lam}(x.\text{ap}(f;x)))); \text{lam}(y.\text{ap}(x;y))) \]

(b) (6 points) Gödel’s T
What mathematical function is defined by the following function in Gödel’s T?

Concrete Syntax: \[ \lambda(n : \text{nat}) \text{rec} n \{ z \Rightarrow s z \mid s(x) \text{ with } y \Rightarrow s(s(y)) \} \]
ABT Syntax: \[ \text{lam}[\text{nat}]\{n. \text{rec}(s(z);x. y.s(s(y)));n) \}

\[ f(n) = 3(n) + 1 \]

(c) (6 points) PCF
Explain in a sentence or two why you should expect that a solution to the traveling salesman problem is definable in PCF. (You need not give any details of what that solution would look like.)

Solution: A graph and a path can both be represented as natural numbers via Gödel encoding, so TSP can be understood as a function from \text{nat} to \text{nat}. By Church’s Law this function is definable in PCF.
Modernized Algol

In the questions on this page, you can use any PCF (with sums, products, unit, and void) syntax you want, along with the expression \texttt{cmd}(m), but from the Algol side you may \textbf{only} use the five constructs of core Algol:

\[
\begin{align*}
  m & ::= \texttt{ret}(e) & \text{ret } e \\
  & | \texttt{bnd}(e;x.m) & \texttt{bnd } x \leftarrow e; m \\
  & | \texttt{set}[a] (e) & a := e \\
  & | \texttt{get}[a] & @a \\
  & | \texttt{dcl}(e;a.m) & \texttt{dcl } a := e \text{ in } m
\end{align*}
\]

as well as the derived \texttt{do}(e) = \texttt{bnd}(e;x.\texttt{ret}(x)) notation. You can use the concrete syntax (it’s preferable!), but do not use the “magic curly braces” syntax.

(d) (6 points) Because both \texttt{unit} and \texttt{unit} + \texttt{unit} are mobile types, we can define a \texttt{while} loop in Algol with the following typing rule:

\[
\frac{
\begin{align*}
  \Gamma \vdash \Sigma m_1 \sim \texttt{unit} + \texttt{unit} \\
  \Gamma \vdash \Sigma m_2 \sim \texttt{unit}
\end{align*}
}{
\Gamma \vdash \Sigma \texttt{while}(m_1) m_2 \sim \texttt{unit}
}\]

If the loop guard returns \texttt{tt} = \texttt{inl}\langle \texttt{unit}, \texttt{unit}\rangle, then the loop body should run and the process should repeat, but if the loop guard returns \texttt{ff} = \texttt{inr}\langle \texttt{unit}, \texttt{unit}\rangle, then the loop should terminate.

Define \texttt{while} loops in terms of the Algol commands \texttt{ret}, \texttt{bnd}, \texttt{set}, \texttt{get}, \texttt{dcl}, \texttt{do}, the expression \texttt{cmd}(m), and any other PCF constructs you need. (You won’t need all six Algol constructs.)

```plaintext
while(m_1) m_2 =

   do (fix loop : cmd unit is
       cmd (bnd guard <- cmd m_1;
           do (case guard of
               { inl _ => cmd (bnd _ <- cmd m_2; do loop)
                | inr _ => cmd (ret <>) }))
```
(e) (6 points) Using this while loop, the six Algol constructs on the previous page, and any
PCF constructs you need, translate this C/C0/Java/pseudo code block into a command
in Modernized Algol.

You can just write 5 instead of $s(s(s(s(z))))$ and you can write 12 instead of writing
$s(s(s(s(s(s(s(s(s(s(z))))))))))$. You can also assume you have an appropriate less-
than-or-equal function

$$leq : \text{naf} \rightarrow \text{naf} \rightarrow (\text{unit} + \text{unit})$$

```plaintext
{ 
    int a = 5;
    while (a <= 12) {
        a = a + 3;
    }
    return a;
}

dcl a := 5 in
bnd _ <- cmd (while (bnd x <- cmd @a; ret (leq x 12))
    (bnd x <- cmd @a;
     bnd _ <- cmd (a := s(s(x)));
     ret <>));
@a
```
Question 2 [40]: Streams

In this question, we will consider *infinite streams of natural numbers*. Streams have a lot in common with the defined-by-elimination sets that we discussed in class.

We’ll start with a concrete presentation of streams that we could add to PCF with sums, products, unit, and void:

z Syntax

\[ e ::= \ldots | \text{hd}(e) | \text{tl}(e) | \text{gen}(e;x.e_h;y.e_t) \]

Statics

\[
\begin{align*}
\Gamma \vdash e : \text{stream} & \quad \Gamma \vdash e : \text{stream} & \quad \Gamma \vdash e : \tau & \quad \Gamma, x : \tau \vdash e_h : \text{nat} & \quad \Gamma, y : \tau \vdash e_t : \tau \\
\Gamma \vdash \text{hd}(e) : \text{nat} & \quad \Gamma \vdash \text{tl}(e) : \text{stream} & \quad \Gamma \vdash \text{gen}(e;x.e_h;y.e_t) : \text{stream}
\end{align*}
\]

Dynamics

\[
\begin{align*}
\text{val} & \quad \text{val} & \quad e \mapsto e' & \quad e \mapsto e' \\
\text{gen}(e;x.e_h;x.e_t) \vdash & \quad \text{gen}(e;x.e_h;y.e_t) \vdash e' \quad \text{gen}(e';x.e_h;y.e_t) & \quad \text{gen}(e;x.e_h;y.e_t) \vdash [e/x]e_h \\
\text{hd}(e) \mapsto \text{hd}(e') & \quad \text{tl}(e) \mapsto \text{tl}(e') & \quad \text{tl}(\text{gen}(e;x.e_h;y.e_t)) \mapsto [e/y]e_t \quad \text{gen}(e;y.e_t) \vdash [e/y]e_t \quad \text{gen}(e;x.e_h;y.e_t)
\end{align*}
\]

The introduction form for streams, \( \text{gen}(e;x.e_h;y.e_t) \), contains a expression \( e \) of some arbitrary type \( \tau \), which we call the *generator* of the infinite stream. We can use the generator to calculate the head of the stream by substituting the generator into \( e_h \). Substituting the generator in for \( y \) in \( e_t \) gives us a new generator of the same type; this is how we calculate the tail.

Here are some examples of streams:

\[
\begin{align*}
\text{alldods} & = \text{gen}(z;x.s(x);y.s(s(y))) \quad 1,3,5,\ldots \\
\text{series}(f) & = \text{gen}(z;x.f(x);y.s(y)) \quad f(0), f(1), f(2), f(3),\ldots \\
\text{iter}_1(f) & = \text{gen}(z;x.x;y.f(y)) \quad 0, f(0), f(f(0)), f(f(f(0))),\ldots \\
\text{iter}_2(f) & = \text{gen}((\lambda(x : \text{nat});x);g.g(z);g.(\lambda(x : \text{nat});f(g(x)))) \quad 0, f(0), f(f(0)), f(f(f(0))),\ldots
\end{align*}
\]

(a) (7 points) State (do not prove) the canonical forms lemma pertaining to the type \( \text{stream} \). Your statement must be strong enough that you could use it in a proof of the progress theorem.

**Solution:** For all \( e \), if \( e \text{ val} \) and \( \emptyset \vdash e : \text{stream} \), then there exist \( e', x, e_h, y \), and \( e_t \) such that \( e = \text{gen}(e';x.e_h;y.e_t) \) and \( e' \text{ val} \).

... and there exists \( \tau' \) such that \( \emptyset \vdash e' : \tau', x : \tau' \vdash e_h : \text{nat}, \) and \( y : \tau' \vdash e_t : \tau' \). (This part is not necessary for proving the progress theorem and so can be omitted.)

Note: The theorem also holds and can be used in the proof of progress if we talk about it in an arbitrary non-empty context \( \Gamma \).
(b) (7 points) State the strongest conclusion that is provable from the following premise using only the statics, and state the induction hypothesis that would permit you to justify this conclusion. You don’t need to prove anything; just state the induction hypothesis.

If $\Gamma \vdash gen(e;x.e_h;y.e_t) : \tau$, then

$\tau = \text{stream}$ and there exists a $\tau_g$ such that

- $\Gamma \vdash e : \tau_g$,
- $\Gamma, x : \tau_g \vdash e_h : \text{nat}$, and
- $\Gamma, y : \tau_g \vdash e_t : \tau_g$.

To prove this, you’d use rule induction with $P(\Gamma' \vdash e' : \tau') =$

For all $e$, $x$, $e_h$, $y$, and $e_t$, if $e' = gen(e;x.e_h;y.e_t)$ then $\tau' = \text{stream}$ and there exists a $\tau_g$ such that

- $\Gamma' \vdash e : \tau_g$,
- $\Gamma', x : \tau_g \vdash e_h : \text{nat}$, and
- $\Gamma', y : \tau_g \vdash e_t : \tau_g$.

(c) (8 points) Write a term of type $\text{stream}$ representing the sequence $0, 1, 0, 1, 0, 1, \ldots$

**Hint:** Think about $\text{iter}_1$ and $\text{iter}_2$. Don’t give your answer in terms of the streams on the previous page, though.

$$\text{flipflop} : \text{stream} = gen(z, x.x, y. \text{ifz y } \{ z => s z \mid s _- => z \})$$

(d) (8 points) Write $map : (\text{nat} \rightarrow \text{nat}) \rightarrow \text{stream} \rightarrow \text{stream}$ that, given a function $f$ and a stream $x_1, x_2, x_3, x_4, \ldots$ produces the stream $f(x_1), f(x_2), f(x_3), f(x_4), \ldots$

**Hint:** the generator of the new stream probably shouldn’t have the type nat.

$$map = \lambda(f : \text{nat} \rightarrow \text{nat}) \lambda(str : \text{stream})\ gen(str, x. f(hd x), y. \text{tl} y)$$
(e) (10 points) Recall the statics and dynamics of recursive types:

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

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

\[
\begin{align*}
e \mapsto e' & \quad \text{unfold} \, (e) \mapsto \text{unfold} \, (e')
\end{align*}
\]

\[
\begin{align*}
e \mapsto e & \quad \text{unfold} \, (\text{fold} \, [t. \tau] \, (e)) \mapsto e
\end{align*}
\]

With recursive types, it is possible to define `stream` as the type `\mu t. (unit \to (nat \times t))`. (If we have lazy pairs, then `\mu t. nat \times t` works too, but `\mu t. (unit \to (nat \times t))` works either way.)

Give a translation of stream syntax that gives us expressions with the statics we gave above and which produces the same streams of numbers as the streams we defined explicitly. That is, for the `allodds` stream defined above, your implementation of `gen`, `hd`, and `tl` should have the property that:

- `hd(allodds) \mapsto s(z),`
- `hd(tl(allodds)) \mapsto s(s(z)),`
- `hd(tl(tl(allodds))) \mapsto s(s(s(z))),`
- \ldots and so on.

For full credit, we strongly suggest that you use our suggested template for `gen`.

**Hint:** You’ll want to perform two substitutions in the body of the `gen` definition, substituting for `x` in `eh` and `y` in `et`.

```haskell

\begin{align*}
\text{gen}(e; x.e_h; y.e_t) & : \mu t. (unit \to (nat \times t)) = \\
(\text{fix newstream} : \tau \to \mu t. (unit \to (nat \times t))) \text{ is} \\
\lambda (\text{generator} : \tau) \\
\quad \text{fold} \, [t. \text{unit} \to (\text{nat} * \text{t})] \\
\quad \text{fn} \, (_ : \text{unit}) \\
\quad \quad < [\text{generator}/x] \, \text{eh}, \\
\quad \quad \text{newstream} \, ([\text{generator}/y] \, \text{et}) > \\
\end{align*}
```

\[
\begin{align*}
\text{hd}(e) & : \text{nat} = (\text{unfold}(e) ()) \cdot 1 \\
\text{tl}(e) & : \mu t. (\text{unit} \to (\text{nat} \times t)) = (\text{unfold}(e) ()) \cdot r
\end{align*}
\]
Question 3 [30]: Stack computation

Stacks are lists of natural numbers. In this question, we will deal with a little language that performs operations on stacks.

We sometimes interpret the numbers we store in stacks as numerals, but depending on the stack type associated with the stack, we may also treat them as the runtime tags that must be associated values of sum type in a language like PCF.

Good ol’ nats

\[ n ::= z \mid s(n) \]

Stacks of nats

\[ k ::= \text{emp} \mid \text{cons}(n; k) \]

Familiar types

\[ \tau ::= \text{nat} \mid \tau_1 + \tau_2 \]

Stacks of types

\[ st ::= \text{empty} \mid \text{consty}(\tau; st) \]

We classify stacks with stack types \( st \). The numbers \( z \) and \( s(z) \) can act both as numerals and as tags telling us what the type of the rest of the stack is.

\[
\begin{align*}
\text{emp} : \text{empty} & \quad k : st \quad k : \text{consty}(\tau_1; st) \\
\text{cons}(n; k) : \text{consty}(\text{nat}; st) & \quad k : \text{consty}(\tau_2; st) \\
\text{cons}(z; k) : \text{consty}(\tau_1 + \tau_2; st) & \quad k : \text{consty}(\tau_1 + \tau_2; st)
\end{align*}
\]

Observe that if a stack type starts with \( \tau_1 + \tau_2 \), the numeral on the top of the corresponding stack type acts like a tag that describes the structure of the next value on the stack.

(a) (5 points) Unicity of typing definitely does not hold for stacks. Here’s one derivation that gives a stack typing to the stack \( \text{cons}(s(z); \text{cons}(z; \text{emp})) \)

\[
\begin{align*}
\text{emp} : \text{empty} & \quad k : \text{consty}(\text{nat}; \text{empty}) \\
\text{cons}(z; \text{emp}) : \text{consty}(\text{nat}; \text{empty}) & \quad k : \text{consty}(\tau + \text{nat}; \text{empty})
\end{align*}
\]

Give a derivation which shows that you can give the stack \( \text{cons}(s(z); \text{cons}(z; \text{emp})) \) some other stack type:

\[
\begin{align*}
\text{emp} : \text{empty} & \quad k : \text{consty}(\text{nat}; \text{empty}) \\
\text{cons}(z; \text{emp}) : \text{consty}(\text{nat}; \text{empty}) & \quad k : \text{consty}(\tau + \text{nat}; \text{empty})
\end{align*}
\]

Solution:

\[
\begin{align*}
\text{emp} : \text{empty} & \quad k : \text{consty}(\text{nat}; \text{empty}) \\
\text{cons}(z; \text{emp}) : \text{consty}(\text{nat}; \text{empty}) & \quad k : \text{consty}(\tau + \text{nat}; \text{empty})
\end{align*}
\]

(Any \( \tau \) will do.)
Commands $c$ in this language are instructions that manipulate the stack: either pushing a new constant onto the stack, adding the two numbers on the top of the stack, tagging the value on the top of the stack with $\text{inl}$ or $\text{inr}$, or doing a case analysis on a sum type’s tag.

$$c ::= \text{retn} \mid \text{push}(n; c) \mid \text{plus}(c) \mid \text{inl}(c) \mid \text{inr}(c) \mid \text{case}(c_l; c_r)$$

Stack computations have the form $c \parallel k$, and a final state must have exactly one number on the stack. These are the dynamics for our stack computation language:

- $\text{retn} \parallel (\text{cons}(n; \text{emp})) \rightarrow \text{final}$
- $\text{push}(n; c) \parallel k \rightarrow c \parallel \text{cons}(n; k)$
- $\text{plus}(c) \parallel \text{cons}(n_1; \text{cons}(n_2; k)) \rightarrow c \parallel \text{cons}(n_3; k)$
- $\text{inl}(c) \parallel k \rightarrow c_l \parallel k$
- $\text{inr}(c) \parallel k \rightarrow c_r \parallel k$
- $\text{case}(c_l; c_r) \parallel \text{cons}(z; k) \rightarrow c_l \parallel k$
- $\text{case}(c_l; c_r) \parallel \text{cons}(s(z); k) \rightarrow c_r \parallel k$

(b) (4 points) Give a stack $k$ such that $\text{case}(c_l; c_r) \parallel k$ would be stuck (unable to step but not final), but $\text{plus}(c) \parallel k$ would not be stuck (either final or able to take a step):

**Solution:** $k = \text{cons}(s(s(n_2)); \text{cons}(n_1; st))$ (any $n_1$, $n_2$, and $st$)

(c) (4 points) Give a stack $k$ such that both $\text{case}(c_l; c_r) \parallel k$ and $\text{plus}(c) \parallel k$ would be stuck:

**Solution:** $k = \text{emp}$ or $k = \text{cons}(s(s(n)); \text{emp})$ for any $n$. 
A command can be judged okay relative to a particular stack type that describes the size and contents of the stack. Thus, the statics of our stack computation language are described by the judgment \( c \parallel st \ ok \).

\[
\begin{align*}
\text{retn} & \parallel \text{consty(nat; empty)} \ ok & \text{t-retn} \\
\text{c} & \parallel \text{consty(nat; st)} \ ok & \text{t-push} \\
\text{c} & \parallel \text{consty(nat; st)} \ ok & \text{t-plus} \\
\text{c} & \parallel \text{consty(nat; st)} \ ok & \text{t-inl} \\
\text{c} & \parallel \text{consty(\tau_1 + \tau_2; st)} \ ok & \text{t-case}
\end{align*}
\]

(d) (5 points) The following supposed preservation theorem is not true:

For all \( c, k, \) and \( st \), if

- \( c \parallel st \ ok \),
- \( k : st \), and
- \( c \parallel k \mapsto c' \parallel k' \),

then \( c' \parallel st \ ok \) and \( k' : st \).

Why? (Either explain with a sentence or give a specific counterexample.)

\[\text{Solution:}\]

The stack type can change over the course of a step. For example,

- \( \text{plus(retn)} \parallel \text{consty(nat; consty(nat; empty)) ok} \),
- \( \text{cons(z; cons(z; emp)) : consty(nat; consty(nat; empty))} \), and
- \( \text{plus(retn)} \parallel \text{consty(nat; consty(nat; empty))} \mapsto \text{ret} \parallel \text{cons(z; emp)} \)

but neither of the following hold:

- \( \text{retn} \parallel \text{consty(nat; consty(nat; empty)) ok} \)
- \( \text{cons(z; emp) : consty(nat; consty(nat; empty))} \)

To fix this, the stack type must be allowed to change in the theorem: we need the conclusion to be that there exists a \( st' \) such that \( c' \parallel st' \ ok \) and \( k' : st' \).
(e) (12 points) Here is a reasonable progress theorem for this language:

**Theorem 1 (Progress).** For all $c$, $k$, and $st$, if $k : st$ and $c \parallel st$ ok, then either there exists $k'$ and $c'$ such that $c \parallel k \rightarrow c' \parallel k'$ or $c \parallel k$ final.

We can prove progress by rule induction on the definition of $c \parallel st$ ok, with $\mathcal{P}(c \parallel st$ ok) = “for all $k$, if $k : st$ then either there exists $k'$ and $c'$ such that $c \parallel k \rightarrow c' \parallel k'$ or $c \parallel k$ final.” For the case corresponding to rule $t$-case only, state the applicable portion of the rule induction principle and carefully prove this case of the progress theorem.

You can use the following inversion lemmas without proof:

- If $k : \text{empty}$, then $k = \text{emp}$.
- If $k : \text{consty(nat; st)}$, then there exist $n$ and $k'$ such that $k = \text{cons}(n; k')$ and $k' : st'$.
- If $k : \text{consty(\tau_1 + \tau_2; st)}$, then either
  - there exists $k'$ such that $k = \text{cons}(\tau_1; k')$ and $k' : \text{consty(\tau_1; st)}$, or
  - there exists $k'$ such that $k = \text{cons}(\tau_2; k')$ and $k' : \text{consty(\tau_2; st)}$.

**Rule induction principle relating to $t$-case:**

For all $\tau_1$, $\tau_2$, $st$, $c_1$, and $c_r$, if $\mathcal{P}(c_1 \parallel \text{consty(\tau_1; st) ok})$ and $\mathcal{P}(c_r \parallel \text{consty(\tau_2; st) ok})$, then $\mathcal{P}(\text{case}(c_1; c_r) \parallel \text{consty(\tau_1 + \tau_2; st) ok})$.

As induction hypotheses (which we won’t need to use), we have $\mathcal{P}(c_1 \parallel \text{consty(\tau_1; st) ok})$ and $\mathcal{P}(c_r \parallel \text{consty(\tau_2; st) ok})$.

To show: For all $k$, if $k : \text{consty(\tau_1 + \tau_2; st)}$ then either there exists $k'$ and $c'$ such that $\text{case}(c_1; c_r) \parallel k \rightarrow c' \parallel k'$ or $\text{case}(c_1; c_r) \parallel k$ final.

Let _____ $k$ _____ be fixed and arbitrary.

1) $k : \text{consty(\tau_1 + \tau_2; st)}$ by assumption

We know the state won’t be final; it suffices to show: there exists $k'$ and $c'$ such that $\text{case}(c_1; c_r) \parallel k \rightarrow c' \parallel k'$

By inversion on (1), we have that there exists a $k''$ such that $k = \text{cons}(\tau_1; k'')$ or there exists a $k''$ such that $k = \text{cons}(\tau_2; k'')$.

In the first case, let $k' = k''$ and let $c' = c_1$; the conclusion follows by $s$-casel.

In the second case, let $k' = k''$ and let $c' = c_r$; the conclusion follows by $s$-caser.