15–312: Principles of Programming Languages

Final Examination

May 6, 2014

• There are 18 pages in this examination, comprising 4 questions worth a total of 100 points.
• You may refer to your personal notes and to Practical Foundations of Programming Languages, but not to any other person or source.
• No electronic devices of any kind are allowed unless they have been cleared in advance by the instructor.
• You have 180 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>Timeout</th>
<th>Polymorphism</th>
<th>Algol</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points:</td>
<td>20</td>
<td>30</td>
<td>15</td>
<td>35</td>
<td>100</td>
</tr>
<tr>
<td>Score:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Question 1 [20]: Short Answer

(a) (6 points) Consider a hybrid-typing extension to the parallel language in Homework 5, where the following four rules define the type `dyn` in its entirety:

\[
\begin{align*}
\Gamma \vdash e : \text{dyn} & & \Gamma \vdash e : \text{dyn} \rightarrow \text{dyn} & & \Gamma \vdash e : \text{dyn} \\
\Gamma \vdash e \text{ @ fun} : \text{dyn} \rightarrow \text{dy}n & & \Gamma \vdash \text{fun}! e : \text{dyn} & & \Gamma \vdash \text{fun}? e : \text{bool} \\
\Gamma \vdash e : \text{dyn} & & \Gamma \vdash e \text{ @ seq} : \text{seq} (\text{dyn}) & & \Gamma \vdash e : \text{dyn} \\
\hline
\Gamma \vdash e \text{ @ seq} : \text{seq} (\text{dyn}) & & \Gamma \vdash \text{seq}! e : \text{dyn} & & \Gamma \vdash \text{seq}? e : \text{bool}
\end{align*}
\]

If we instead extended the language from Homework 5 with sums and recursive types, give a recursive type that could be used to implement this `dyn` type:

Given that `dyn` is defined as you described, implement `seq? e` such that, if \(v_1\) and \(v_2\) are appropriately typed values, \(\text{seq}? (\text{seq}! v_1) \rightarrow ^* \text{true}\) and \(\text{seq}? (\text{fun}! v_2) \rightarrow ^* \text{false}\).

(b) (4 points) By the proofs-as-programs principle, a proof of a given proposition corresponds to a term of a given type. For the following to logical statements, state the types that they correspond to by this principle. One example is given.

Both \(A\) and \(B\) are true.

\[
A \times B
\]

If \(A\) and \(B\), then either \(C\) or \(D\).

Either \(A\) is true or \(B\) is false.
(c) (10 points) Here are the (only!) statics and value judgment rules for the introduction forms of a type $\tau_1 \otimes \tau_2$:

\[
\begin{align*}
\Gamma \vdash e_1 : \tau_1 && \Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash \text{foo}(e_1; e_2) : \tau_1 \otimes \tau_2 && \Gamma \vdash \text{bar}(e) : \tau_1 \otimes \tau_2 \\
e_1 \text{ val} && \text{foo}(e_1; e_2) \text{ val} && e \text{ val}
\end{align*}
\]

Give appropriate and deterministic small-step structural dynamics rules for $\text{foo}(e_1; e_2)$ and $\text{bar}(e)$:

Define (an) appropriate elimination form(s) for $\tau_1 \otimes \tau_2$ and give both statics and deterministic dynamics. (There’s more than one reasonable way to do this.) You shouldn’t mention any types besides $\tau_1 \otimes \tau_2$ in the statics you give.
Question 2 [30]: Timeout

In this exercise, we will enrich PCF – strict nats and functions only, no sums or products – with a timeout operator. To do so, we extend PCF’s expressions with the following additional form:

<table>
<thead>
<tr>
<th>Sort</th>
<th>Abstract Form</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expressions</td>
<td>$e ::= \text{timeout}(e_1; e_2; e_3)$</td>
<td>Timeout</td>
</tr>
</tbody>
</table>

In this expression, $e_1$ is a provided upper limit to the number of steps $e_2$ can take to evaluate. If $e_2$ evaluates to a value in fewer than this number of steps, then the overall value of this expression is the value of $e_2$, otherwise it is the value of $e_3$.

(a) (2 points) Write the typing rule for $\text{timeout}(e_1; e_2; e_3)$.

(b) (4 points) The dynamic semantics of $\text{timeout}(e_1; e_2; e_3)$ has four rules. The first one is the following:

$$
\text{timeout}(e_1; e_2; e_3) \Rightarrow \text{timeout}(e_1'; e_2; e_3) \quad ^{\text{ev.to1}}
$$

Define the remaining three. The resulting dynamics should be deterministic; for any given expression, there should be at most one transition rule. Recall that this expression evaluates to the value of $e_2$ if this value can be produced in fewer than $e_1$ steps, and to the value of $e_3$ otherwise.
(c) (4 points) When implementing user interfaces, the ability to delay the evaluation of an expression can be useful. Using `timeout`, define the expression `delay(e_1, e_2)` that introduces `e_1` steps of delay before stepping to `e_2`. You may assume that `e_2` has type `τ`.

In the next two parts, your implementations needn’t be work efficient; it is okay for them to be wildly inefficient if they meet the specification.

(d) (4 points) Using `timeout`, define the expression `earliest(e_1; e_2)` that returns the value of whichever among `e_1` and `e_2` takes the least number of steps to produce a value (it returns the value of `e_1` if they take the same number of steps). Note that one or both arguments may be divergent but your expression should only diverge if both arguments diverge.

If convenient, you may use a call-by-value `let` and/or define auxiliary functions.
(e) (6 points) Using timeout, define the expression \( \text{numsteps}(e) \) that returns the number of steps it takes to evaluate \( e \) (or diverges if \( e \) diverges).

If convenient, you may use a call-by-value \texttt{let} and/or define auxiliary functions.
(f) (10 points) State and rigorously prove the progress theorem for this language, limiting
yourself to the cases that involve timeout. If you need standard lemmas (e.g. canonical
forms, inversion, etc.) you must separately state them yourself and cite them explicitly,
but you do not need to prove them yourself.
Question 3 [15]: Polymorphism

In this question, we will work in an extension of System F with pairs $\tau_1 \times \tau_2$, existential types $\exists t.\tau$, and natural numbers $\text{nat}$.

**Types**

$$\tau ::= t \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \forall t.\tau \mid \exists t.\tau \mid \text{nat}$$

**Expressions**

$$e ::= x \mid \lambda(x : \tau)e \mid e_1(e_2) \mid e \cdot 1 \mid e \cdot x \mid \Lambda(t)e \mid e[\tau] \mid \text{pack } \rho \text{ with } e \text{ as } \exists t.\tau \mid \text{open } e_1 \text{ as } t \text{ with } x : \tau \text{ in } e_2 \mid z \mid s(e) \mid \text{ifz } e\{z \Rightarrow e_1 \mid s(x) \Rightarrow e_2\}$$

We lose some power in our natural numbers because we only provided the $\text{ifz}$ elimination form for natural numbers instead of the recursor, but that won’t matter for this question.

(a) (4 points) Define the Church encoding of option types $\text{opt}(\rho)$ with the following type structure:

```
\Delta, \Gamma \vdash \text{NONE}[\rho] : \text{opt}(\rho) \\
\Delta, \Gamma \vdash e : \rho \\
\Delta, \Gamma \vdash \text{SOME}(e) : \text{opt}(\rho)
```

$$\Delta, \Gamma \vdash \text{ocase } e \{\text{NONE } \rightarrow e_1 \mid \text{SOME}(x) \Rightarrow e_2\} : \tau$$

And the following equational properties:

$$\text{ocase NONE}[\tau] \{\text{NONE } \rightarrow e_1 \mid \text{SOME}(x) \Rightarrow e_2\} \equiv e_1$$

$$\text{ocase SOME}(e) \{\text{NONE } \rightarrow e_1 \mid \text{SOME}(x) \Rightarrow e_2\} \equiv [e/x]e_2$$

In the translations below, you may assume $e$ has type $\rho$, that $e_o$ has type $\text{opt}(\rho)$ as you defined it, that $e_1$ has type $\tau$, and that if $x$ is a variable of type $\rho$, $e_2$ has type $\tau$.

<table>
<thead>
<tr>
<th>$\text{opt}(\rho)$</th>
<th>$\triangleq$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{NONE}[\rho]$</td>
<td>$\triangleq$</td>
</tr>
<tr>
<td>$\text{SOME}(e)$</td>
<td>$\triangleq$</td>
</tr>
<tr>
<td>$\text{ocase } e_o {\text{NONE } \rightarrow e_1 \mid \text{SOME}(x) \Rightarrow e_2}$</td>
<td>$\triangleq$</td>
</tr>
</tbody>
</table>
This Church encoding is just one way of implementing the option type. If we want to write code that can work with multiple representations, we can use existential types. Recall the rules for introducing and eliminating existential types:

\[
\frac{\Delta, \Gamma \vdash \rho \text{ type} \quad \Delta, \tau \text{ type} \quad \Delta, \Gamma \vdash e : [\rho/t]\tau}{\Delta, \Gamma \vdash \text{pack } \rho \text{ with } e \text{ as } \exists \!\! \exists t.\tau} \\
\frac{\Delta, \Gamma \vdash e_1 : \exists t.\tau \quad \Delta, \tau \text{ type} \quad \Delta, \Gamma \vdash x : \tau \vdash e_2 : \tau_2}{\Delta, \Gamma \vdash \text{open } e_1 \text{ as } t \text{ with } x : \tau \in e_2 : \tau_2}
\]

The type we will use for a packaged existential type of optional \( \text{nat} \) values is

\[
\exists t. (t \times (\text{nat} \to t)) \times (\forall p. t \to p \to (\text{nat} \to p) \to p)
\]

(b) (5 points) Unpack an unknown implementation \textbf{let}-bound as \textit{impl} and assign the pieces \textit{none}, \textit{some}, and \textit{ocase} correctly:

\begin{verbatim}
let impl : \exists t. (t \times (\text{nat} \to t)) \times (\forall p. t \to p \to (\text{nat} \to p) \to p) = (omitted)

let none =

let some =

let ocase =
...
\end{verbatim}

(c) (3 points) Implement this type, using types and syntax from part (a):

\begin{verbatim}
pack opt(\text{nat}) with

as \exists t. (t \times (\text{nat} \to t)) \times (\forall p. t \to p \to (\text{nat} \to p) \to p)
\end{verbatim}

(d) (3 points) Implement this type using only natural numbers:

\begin{verbatim}
pack nat with

as \exists t. (t \times (\text{nat} \to t)) \times (\forall p. t \to p \to (\text{nat} \to p) \to p)
\end{verbatim}
Question 4 [35]: Algol

In this question we will develop an abstract machine semantics for Algol commands \( s \mapsto s' \), while keeping the usual structural dynamics \( e \mapsto e' \) for expressions. There are two states of this machine: 

\[
\begin{align*}
\Sigma \vdash k : \tau & \quad \emptyset \vdash \Sigma m \sim \tau \\
(k \triangleright m) \triangleright & \quad \Sigma k : \tau \quad \emptyset \vdash \Sigma v : \tau \quad v \text{ val}_\Sigma \\
& \quad (k \preceq v) \triangleright
\end{align*}
\]

Like in lecture (but unlike in the homework assignment on parallelism), we will be unconcerned with the ultimate return type of states and stacks, so \( k : \tau \) means that \( k \) is a stack that may have values of type \( \tau \) returned to it.

\[
\begin{align*}
\vdash \Sigma e : \tau \quad \vdash \Sigma f : \tau' \\
\vdash \Sigma (k; f) : \tau
\end{align*}
\]

If we just care about the Algol commands \( \text{ret } e \) and \( \text{bnd } x \leftarrow e; m \), the dynamics and statics of commands are relatively simple. We only have to worry about one frame!

\[
\begin{align*}
\Sigma e & \mapsto e' \\
(k \triangleright \text{ret } e) & \mapsto (k \triangleright \text{ret } e') \\
\Sigma v \text{ val}_\Sigma & \quad \Sigma (k \triangleright \text{ret } v) & \mapsto (k \preceq v) \\
(k \triangleright \text{bnd } x \leftarrow e; m) & \mapsto (k \triangleright \text{bnd } x \leftarrow e'; m) \\
(k \triangleright \text{cmd}(m'); m) & \mapsto (k; \begin{array}{c} \text{(omitted)} \end{array} \triangleright m')
\end{align*}
\]

(a) (6 points) State what the frame that was omitted above should be by giving the remaining dynamic semantics rule (where a value is returned to the omitted frame) and the static semantics rule for that omitted frame.
Given a \texttt{bool} and \texttt{unit} type in the expression language, we can add while loops to the language as a primitive construct:

$$
\Gamma \vdash_{\Sigma} m_1 \sim \texttt{bool} \quad \Gamma \vdash_{\Sigma} m_2 \sim \texttt{unit} \\
\Gamma \vdash_{\Sigma} \texttt{while} \ m_1 \ m_2 \sim \texttt{unit}
$$

We add two additional forms of frame, \texttt{whilecond}(m_1;m_2) and \texttt{whilebody}(m_1;m_2), to present the dynamics:

$$( k \triangleright \texttt{while} \ m_1 \ m_2 \biggm{\downarrow} ) \xrightarrow{\Sigma} ( k \triangleright \texttt{whilecond}(m_1;m_2) \biggm{\downarrow} m_1 \biggm{\uparrow} )$$

$$( k \triangleright \texttt{whilecond}(m_1;m_2) \biggm{\downarrow} \texttt{false} \biggm{\downarrow} ) \xrightarrow{\Sigma} ( k \triangleright ( ) )$$

$$( k \triangleright \texttt{whilecond}(m_1;m_2) \biggm{\downarrow} \texttt{true} \biggm{\downarrow} ) \xrightarrow{\Sigma} ( k \triangleright \texttt{whilebody}(m_1;m_2) \biggm{\downarrow} m_2 \biggm{\uparrow} )$$

$$( k \triangleright \texttt{whilebody}(m_1;m_2) \biggm{\downarrow} ( ) \biggm{\downarrow} ) \xrightarrow{\Sigma} ( k \triangleright \texttt{whilecond}(m_1;m_2) \biggm{\downarrow} m_1 \biggm{\uparrow} )$$

(b) (4 points) Give appropriate statics for these two frames:

(c) (4 points) Consider changing the statics we initially gave for \texttt{while} to this:

$$
\Gamma \vdash_{\Sigma} m_1 \sim \texttt{bool} \quad \Gamma \vdash_{\Sigma} m_2 \sim \tau \\
\Gamma \vdash_{\Sigma} \texttt{while} \ m_1 \ m_2 \sim \texttt{unit}
$$

What changes, if any, would we need to make to (our) dynamics or (your) statics in order to preserve progress and preservation?
One reason we might want to make while loops primitive is so that we can introduce the additional commands break and continue. If, during evaluation, we reach a break or continue, we immediately exit or restart (respectively) the innermost while loop whose body contains that break or continue. We will implement these commands in terms of two new abstract machine states, \( s ::= \ldots | ( k \downarrow \text{break} ) | ( k \downarrow \text{continue} ) \).

\[
\begin{align*}
\Gamma \vdash \Sigma k : \tau & \quad & \Gamma \vdash \Sigma k : \tau \\
( k \downarrow \text{break} ) \text{ ok}_\Sigma & \quad & ( k \downarrow \text{continue} ) \text{ ok}_\Sigma \\
\end{align*}
\]

A loop will only catch a break or continue if it is evaluating the loop body, so whilecond frames are passed through:

\[
\begin{align*}
( k ; \text{whilecond}(m_1; m_2) \downarrow \text{break} ) & \rightsquigarrow \Sigma ( k \downarrow \text{break} ) \\
( k ; \text{whilecond}(m_1; m_2) \downarrow \text{continue} ) & \rightsquigarrow \Sigma ( k \downarrow \text{continue} ) \\
\end{align*}
\]

(d) (8 points) Give the remaining dynamic semantics for break and continue. Make sure to account for the frame you defined in part (a).

(e) (4 points) Do the rules for \(( k \downarrow \text{break} ) \text{ ok}_\Sigma\) and \(( k \downarrow \text{continue} ) \text{ ok}_\Sigma\) require the \( k : \tau\) premise? Why or why not?
In Concurrent Algol, there was one really annoying judgment:

\[
\frac{\Sigma \vdash m \nu \Sigma' \{m' \parallel P\}}{}
\]

One way to diagnose the complexity of this judgment was that, in in-lecture and in-PFPL formulation of concurrent Algol, a single step of the command \( m \) might do one of five things:

- Synchronize on an evaluated event, a value of type \( \text{event}(\tau) \), with the action \( \alpha \) using the judgment \( e \xrightarrow{\alpha} m \), and produce a new command:

\[
\frac{\Sigma \vdash e \nu \Sigma \{m \parallel \text{stop}\}}{\Sigma \vdash \text{sync}(e) \nu \Sigma' \{m' \parallel P\}}
\]

- Spawn a new process:

\[
\text{spawn(cmd}(m)) \xrightarrow{\Sigma} \nu \{\text{ret } \emptyset \parallel \text{proc}(m)\}
\]

- Generate a new channel:

\[
\text{newchn}[\tau] \xrightarrow{\Sigma} \nu \{a \xrightarrow{\tau} \text{ret } \text{chan}[a] \parallel \text{stop}\}
\]

- It may take an uneventful step, doing none of the three previous actions:

\[
\frac{e \xrightarrow{\Sigma} e'}{\Sigma \vdash \text{ret } e \xrightarrow{\Sigma} \nu \{e'/x \parallel m \parallel \text{stop}\}}
\]

\[
\frac{e \xrightarrow{\Sigma} e'}{\Sigma \vdash \text{bnd } x \leftarrow e; m \xrightarrow{\Sigma} \nu \{\text{bnd } x \leftarrow e'; m \parallel \text{stop}\}}
\]

\[
\frac{e \xrightarrow{\Sigma} e'}{\Sigma \vdash \text{sync}(e) \xrightarrow{\Sigma} \nu \{\text{sync}(e') \parallel \text{stop}\}}
\]

- It may evaluate a sequenced command \( \text{bnd } x \leftarrow \text{cmd}(m); m' \), meaning that it has to deal with any of the four above possibilities happening in the first command:

\[
\frac{\Sigma \vdash m \nu \Sigma' \{m' \parallel P\}}{\Sigma \vdash \text{bnd } x \leftarrow \text{cmd}(m); m'' \xrightarrow{\Sigma} \nu \Sigma' \{\text{bnd } x \leftarrow \text{cmd}(m'); m'' \parallel P\}}
\]

This judgment then interacted with the transition rules for the process calculus (the definition of the judgment \( P \xrightarrow{\Sigma} P' \)) at a single point:

\[
\frac{\Sigma \vdash m \nu \Sigma' \{m' \parallel P\}}{\Sigma \vdash \text{proc}(m) \xrightarrow{\Sigma} \nu \Sigma' \{\text{proc}(m') \parallel P\}}
\]
By using control stacks, we can get rid of the judgment $m \xrightarrow{\alpha} \nu \Sigma' \{ m' \parallel P \}$ and take advantage of the fact that, while many different things might happen when we evaluate a command, at most one interesting thing (spawning a process, synchronizing on an event, creating a channel) ever happens at a time.

Instead of atomic processes having the form $\text{proc}(m)$, they will have the form $\text{proc}(s)$ in our reformulation: that is, either the form $\text{proc}(k \triangleright m)$ or $\text{proc}(k \triangleleft \nu)$.

$$
\begin{align*}
s \xrightarrow{\Sigma} s' & \quad \text{proc}(s) \xrightarrow{\Sigma} \text{proc}(s') \\
\text{proc}(\epsilon \triangleleft \nu) & \xrightarrow{\Sigma} \text{stop}
\end{align*}
$$

$$
\begin{align*}
P_1 \xrightarrow{\alpha} P_1' & \quad \frac{P \xrightarrow{\alpha} P' \vdash \Sigma \alpha \text{ action}}{\nu a \sim \tau \{ P \} \xrightarrow{\alpha} \nu a \sim \tau \{ P \}}
\end{align*}
$$

Each Concurrent Algol feature, like $\text{sync}(e)$, can then be defined with a mix of abstract machine rules that derive judgments of the form $s \xrightarrow{\Sigma} s'$ and process calculus rules that derive judgments of the form $\text{proc}(s) \xrightarrow{\alpha} P$.

$$
\begin{align*}
e \xrightarrow{\Sigma} e' & \quad (k \triangleright \text{sync}(e)) \xrightarrow{\Sigma} (k \triangleright \text{sync}(e')) \\
\text{proc}(k \triangleright \text{sync}(e)) & \xrightarrow{\alpha} \text{proc}(k \triangleright m)
\end{align*}
$$

The abstract machine dynamics we already gave for $\text{ret} e$ and $\text{bnd} x \leftarrow e; m$ are sufficient; we don’t need any additional rules for these features.

(f) (9 points) Give the dynamics of $\text{spawn}(e)$ and $\text{newchn}[\tau]$.  

---

15–312 Final 15 of 18 May 6, 2014
Scratch Work:
Scratch Work:
Scratch Work: