15–312: Principles of Programming Languages

Final Examination

May 9, 2017 1pm to 4pm

- There are 16 pages in this examination, comprising 5 questions worth a total of 120 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 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: ____________________

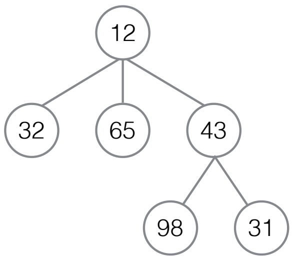
Question:	Trees	Cont	FF	Exc	Space	Total
Points:	15	10	30	30	35	120
Score:						

Question 1 [15]: Rose Trees in System F

Recall the definition of **System F**. For your convenience we repeat the syntax here.

$$\begin{array}{llll} \textsf{type} & \tau & ::= & t \\ & & & \tau_1 \to \tau_2 \\ & & & \forall(t.\tau) \end{array} \qquad \begin{array}{llll} \textsf{exp} & e & ::= & x \\ & & & \lambda\,(x:\tau)\,e \\ & & & e_1(e_2) \\ & & & \Lambda(x)\,e \\ & & & e[\tau] \end{array}$$

A rose tree is a tree in which each node has a variable and unbounded number of children.



Rose trees with elements of type α can be implemented with the following recursive type.

$$R(\alpha) = \texttt{rec}\,t\,\texttt{is}\,\alpha \times \texttt{rec}\,u\,\texttt{is}\,\texttt{unit} + (t \times u)$$

(a) (6 points) Translate the type $R(\alpha)$ as a polymorphic type $\overline{R}$ of the form $\forall(\alpha.\tau)$ in **System F**.

Hint: First encode the inner recursive type in System F. Recall Church encodings.

Solution: $\forall(\alpha.\forall(t.(\alpha \to \forall(u.u \to (t \to u \to u) \to u) \to t) \to t))$

(b) You will now define the singleton tree r_0 that consists of a single node 0 as a term of type $\overline{R}[\texttt{nat}]$ in **System F**.

i. (3 points) Define the term $nil[t]$, that represents the empty list of element type t.

Solution:

$$nil[\mathrm{t}] \quad \triangleq \quad \Lambda(u)\,\lambda\,(x:u)\,\lambda\,(f:t \to u \to u)\,x$$

(Question continues on the next page)

ii. (6 points) Use $nil[t]$ for an appropriate t to define the singleton tree r_0 that consists of node 0 as a term of type $\overline{R}[\mathtt{nat}]$ in **System F**.
You can use the natural number 0 in your term as an abbreviation for the actual encoding of natural numbers in **System F**.

Solution:

$$r_0 \triangleq \Lambda(t)\,\lambda\,(f : \mathtt{nat} \to \forall(u.u \to (t \to u \to u) \to u) \to t)\, f(0)(nil[t])$$

Question 2 [10]: Continuations

For this question, you will work with a total language with continuations, products, and sums. The syntax of the language is given below and the static and dynamic semantics is defined as in lecture.

type	τ	$::=$	$\texttt{cont}(\tau)$	exp	e	$::=$	$\texttt{letcc}\{\tau\}(x.e)$
			$\texttt{nat}$				$\texttt{throw}\{\tau\}(e_1; e_2)$
			$\tau_1 \to \tau_2$				$\texttt{cont}(k)$
			$\tau_1 + \tau_2$				$\lambda\,(x : \tau)\, e$
			$\tau_1 \times \tau_2$				$e_1(e_2)$
							$\langle e_1, e_2 \rangle$
							$e \cdot \texttt{l}$
							$e \cdot \texttt{r}$
							$\texttt{case}\, e\, \{\texttt{l} \cdot x \hookrightarrow e_1 \mid \texttt{r} \cdot y \hookrightarrow e_2\}$
							$\texttt{l} \cdot e$
							$\texttt{r} \cdot e$

As a reminder, the static semantics for continuations is given by the following rules:

$$\frac{\Gamma, x : \tau\ \texttt{cont} \vdash e : \tau}{\Gamma \vdash \texttt{letcc}\{\tau\}(x.e) : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_1\ \texttt{cont}}{\Gamma \vdash \texttt{throw}\{\tau\}(e_1; e_2) : \tau}$$

Exhibit terms of the following types.

(a) (4 points) $(\texttt{cont}(\tau_2) \to \texttt{cont}(\tau_1)) \to \tau_1 \to \tau_2$

Solution:

$$\lambda\,(f : \texttt{cont}(\tau_2) \to \texttt{cont}(\tau_1))\,\lambda\,(x : \tau_1)\,\texttt{letcc}\{\tau_2\}(y.\texttt{throw}\{\tau_2\}(x; f(y)))$$

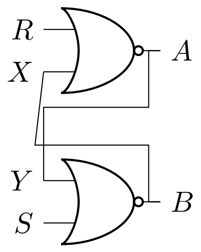
(b) (6 points) $\texttt{cont}(\texttt{cont}(\tau_1) + \texttt{cont}(\tau_2)) \to (\tau_1 \times \tau_2)$

Solution:

$\tau \triangleq \texttt{cont}(\tau_1) + \texttt{cont}(\tau_2)$
$e_1 \triangleq \texttt{letcc}\{\tau_1\}(y.\texttt{throw}\{\tau_1\}(\texttt{l} \cdot y; x))$
$e_2 \triangleq \texttt{letcc}\{\tau_2\}(z.\texttt{throw}\{\tau_2\}(\texttt{r} \cdot z; x))$
$\lambda\,(x : \texttt{cont}(\tau))\,\langle e_1, e_2 \rangle$

Question 3 [30]: Flip Flops

The following diagram depicts an *RS Latch*, a logic gate comprising two cross-coupled *nor* gates whose inputs and outputs are all booleans. The A output of the latch is governed by the R and S inputs, which may not be both false or both true at the same time. If the R input is set to true, and the S input is set to false, then the A output is driven to false; the latch is reset. If, on the other hand, the S input is set to true, and the R input to false, then the A output is driven to true; the latch is set. The B output behaves exactly the opposite way around.



In this question you are to develop an implementation of an RS latch in **PCF** extended with booleans and products, as they are defined in *PFPL*.

The purpose of the exercise is to test your knowledge of self-reference (fixed points), laziness and eagerness, and your ability to define an extension to this language to account for the expected behavior of the latch. Laziness is defined to mean that all constructors are evaluated lazily, and that function applications are call-by-name; eagerness is defined to mean that all constructors are eager, and that function applications are call-by-value.

Please use pattern matching notation for functions whose domain is a product type, it will make your solution much clearer.

(a) (4 points) Give a definition of the function `nor` of type $\texttt{bool} \times \texttt{bool} \to \texttt{bool}$ such that `nor` exhibits under *either* a lazy or eager interpretation:

$$\begin{aligned}
\texttt{nor}(\langle \texttt{false}, \texttt{false}\rangle) &\longmapsto^* \texttt{true}\\
\texttt{nor}(\langle \texttt{false}, \texttt{true}\rangle) &\longmapsto^* \texttt{false}\\
\texttt{nor}(\langle \texttt{true}, \texttt{false}\rangle) &\longmapsto^* \texttt{false}\\
\texttt{nor}(\langle \texttt{true}, \texttt{true}\rangle) &\longmapsto^* \texttt{false}
\end{aligned}$$

There are essentially two possible answers, which differ in the order in which the inputs are considered; give *both* implementations, called $\texttt{nor}_1$ and $\texttt{nor}_2$, using the notation of Section 11.3 of *PFPL*:

Solution:

$$\begin{aligned}
\texttt{nor}_1 &\triangleq \lambda\,(\langle x, y\rangle : \texttt{bool} \times \texttt{bool})\,\texttt{if}\,x\,\texttt{then}\,\texttt{false}\,\texttt{else}\,\texttt{if}\,y\,\texttt{then}\,\texttt{false}\,\texttt{else}\,\texttt{true}\\
\texttt{nor}_2 &\triangleq \lambda\,(\langle x, y\rangle : \texttt{bool} \times \texttt{bool})\,\texttt{if}\,y\,\texttt{then}\,\texttt{false}\,\texttt{else}\,\texttt{if}\,x\,\texttt{then}\,\texttt{false}\,\texttt{else}\,\texttt{true}.
\end{aligned}$$

(Question continues on the next page)

(b) (4 points) By definition $\texttt{nor}_1$ and $\texttt{nor}_2$ behave the same on values, and so in an *eager* setting there are no inputs that would cause one to behave differently from the other. But that is not true in a *lazy* setting. Give a pair of inputs on which $\texttt{nor}_1$ behaves differently from $\texttt{nor}_2$ in a lazy setting. *Hint*: one argument must be a divergent computation; do you see why?

Solution: Let `diverge` be $\texttt{fix}\,x\,\texttt{is}\,x$, which diverges when evaluated.

$$\texttt{nor}_1(\langle\texttt{true},\texttt{diverge}\rangle) \longmapsto^* \texttt{false}$$
$$\texttt{nor}_2(\langle\texttt{true},\texttt{diverge}\rangle) \uparrow$$

And symmetrically for the other choice of evaluation order.

(c) (4 points) Give a definition of the function `rsl` of type $\texttt{bool} \times \texttt{bool} \to \texttt{bool} \times \texttt{bool}$ whose inputs are R and S, respectively, and whose outputs are A and B, respectively. Define your answer using either version of the `nor` function, $\texttt{nor}_1$ or $\texttt{nor}_2$, but be clear about which you are using. *Hint*: your code should implement the wiring diagram given above, ensuring that the outputs are properly "fed back" as inputs.

Solution:

$$\texttt{rsl} \triangleq \lambda\,(\langle r,s\rangle : \texttt{bool}\times\texttt{bool})\,\texttt{fix}\,\langle x,y\rangle\,\texttt{is}\,\langle \texttt{nor}_1(\langle r,x\rangle),\texttt{nor}_1(\langle y,s\rangle)\rangle.$$

The choice of `nor` function is immaterial; neither works properly.

(d) (4 points) Argue that under an *eager* evaluation strategy, your definition of `rsl` diverges for *any* input values, and hence cannot be considered a valid implementation of the RS latch.

Solution: The self-reference will be unrolled infinitely because of eager evaluation of the pairs.

(e) (4 points) Show that under lazy evaluation, your definition of `rsl` behaves properly on some *value* inputs, but not others. More precisely, give one pair of boolean values v_1 and v_2 such that $\texttt{rsl}(\langle v_1, v_2\rangle)\cdot\texttt{l}$ evaluates properly, and another pair v_1' and v_2' such that $\texttt{rsl}(\langle v_1', v_2'\rangle)\cdot\texttt{l}$ evaluates improperly. Notice that we are only considering the A output of the latch, which is the first projection.

Solution: Assume $\texttt{nor}_1$ is used for both components of the latch; the other case is symmetric. Then $v_1 = \texttt{true}$ and $v_2 = \texttt{false}$ leads to the projected application terminating with `false`, as expected. But $v_1' = \texttt{false}$ and $v_2' = \texttt{true}$ leads to divergence. Notice that inspection of the A lead only is important; if we inspect both the A and the B leads, we get divergence for one or the other in these cases.

(f) (10 points) It can be proved, with considerable difficulty, that there is *no solution* to the `rsl` problem in either eager or lazy **PCF** enriched with booleans and products. We must instead enrich **PCF** with a special "built-in" form of `nor` whose evaluation is *non-deterministic* in that there can be more than one next state of evaluation for a given use of `nor`. Give a non-deterministic dynamics for $\texttt{nor}(e_1; e_2)$ such your implementation of `rsl` using this form of `nor` will behave properly on all inputs. *Hint*: your answer will require five rules.

Solution: The evaluation is *non-deterministic*, sometimes called *parallel*, but not in the sense of *PFPL*. The arguments are interleaved, and evaluation stops as early as possible, without preference to which argument achieves `true` first.

$$\frac{e_1 \longmapsto e_1'}{\texttt{nor}(e_1; e_2) \longmapsto \texttt{nor}(e_1'; e_2)}$$

$$\frac{e_2 \longmapsto e_2'}{\texttt{nor}(e_1; e_2) \longmapsto \texttt{nor}(e_1; e_2')}$$

$$\frac{}{\texttt{nor}(\texttt{false}; \texttt{false}) \longmapsto \texttt{true}}$$

$$\frac{}{\texttt{nor}(\texttt{true}; e_2) \longmapsto \texttt{false}}$$

$$\frac{}{\texttt{nor}(e_1; \texttt{true}) \longmapsto \texttt{false}}$$

Question 4 [30]: Exceptions in Algol

Recall from class that an exception mechanism consists of two separable parts, the *control* mechanism, which allows for transfers from the raiser to the handler, and the *data* mechanism, which passes a value from the raiser to the handler. The control aspect was studied in the context of a functional expression language; the data aspect, which we isolated as dynamic classification, was studied in the context of an imperative language. In this question you are to consider an integrated account of both the control and data aspects of exceptions within the command language of Modernized Algol, taking into account both scoped and free assignables.

The syntax of commands in extended **MA** is defined as follows:

Cmd	m	$::=$	$\mathtt{dcl}(e; a.m)$	$\mathtt{dcl}\, a := e \,\mathtt{in}\, m$	new assignable
		$\mid$	$\mathtt{get}[a]$	$@\, a$	fetch
		$\mid$	$\mathtt{set}[a](e)$	$a := e$	assign
		$\mid$	$\mathtt{ret}(e)$	$\mathtt{ret}\, e$	return
		$\mid$	$\mathtt{bndow}(e; x_1.m_1; x_2.m_2)$	$\mathtt{bndow}\, x_1 \hookrightarrow e \,\mathtt{in}\, m_1 \,\mathtt{ow}\, x_2 \hookrightarrow m_2$	sequence/handle
		$\mid$	$\mathtt{cls}\{\tau\}(b.m)$	$\mathtt{cls}\, b \,\mathtt{of}\, \tau \,\mathtt{in}\, m$	new class
		$\mid$	$\mathtt{raise}(e)$	$\mathtt{raise}(e)$	raise exception

The first four are as usual, but the sequencing construct has been extended to include both normal and exceptional ("otherwise") returns, with the following statics:

$$\frac{\Gamma \vdash_\Sigma e : \tau\,\mathtt{cmd} \quad \Gamma, x_1 : \tau \vdash_\Sigma m_1 \mathrel{\dot{\sim}} \tau' \quad \Gamma, x_2 : \mathtt{clsfd} \vdash_\Sigma m_2 \mathrel{\dot{\sim}} \tau'}{\Gamma \vdash_\Sigma \mathtt{bndow}(e; x_1.m_1; x_2.m_2) \mathrel{\dot{\sim}} \tau'}$$

The usual sequencing construct, $\mathtt{bnd}\, x \leftarrow e\,;\, m$, may be defined as $\mathtt{bndow}\, x \hookrightarrow e \,\mathtt{in}\, m \,\mathtt{ow}\, y \hookrightarrow \mathtt{raise}(y)$, which just propagates exceptions. Analogously, an exception handler command, $\mathtt{try}\, e \,\mathtt{ow}\, x \hookrightarrow m$, may be defined dually by $\mathtt{bndow}\, x \hookrightarrow e \,\mathtt{in}\, \mathtt{ret}(x) \,\mathtt{ow}\, y \hookrightarrow m$, which propagates normal returns.

The declaration $\mathtt{cls}\{\tau\}(b.m)$ introduces a new class (aka "exception"), b, with associated value of type τ for use within the command m, analogously to the declaration of assignables. Classes are used to build elements of type $\mathtt{clsfd}$, which is the type of exception values. The command $\mathtt{raise}(e)$ raises an exception with associated value e of type $\mathtt{clsfd}$. This value is passed to the exception return of the nearest enclosing bind, which may propagate it further, or dispatch on the class using the machinery of dynamic classification given in Chapter 33 of *PFPL*, the syntax of which is given by the following grammar:

Typ	τ	$::=$	$\mathtt{clsfd}$	$\mathtt{clsfd}$	classified
Exp	e	$::=$	$\mathtt{in}[b](e)$	$b \cdot e$	instance
			$\mathtt{isin}[b](e; x.e_1; e_2)$	$\mathtt{match}\, e \,\mathtt{as}\, b \cdot x \hookrightarrow e_1 \,\mathtt{ow} \hookrightarrow e_2$	comparison

You are to consider a *scoped* dynamics for classes/exceptions in the sense that no class/exception is allowed to escape the scope of its declaration by any means. The scoped dynamics for commands in **MA** has transitions the form $m \parallel \mu \underset{\Sigma}{\longmapsto} m' \parallel \mu'$, and the scoped dynamics for expressions has transitions of the form $e \underset{\Sigma}{\longmapsto} e'$.

(Question continues on the next page)

(a) (8 points) Give the scoped dynamics for the class declaration command, following the pattern of assignable declaration. Your solution will require *three* rules to account for computing within the scope of the declaration and for exiting the scope of the declaration.

Solution:

$$\frac{m \parallel \mu \underset{\Sigma, b\sim\tau}{\longmapsto} m' \parallel \mu'}{\texttt{cls}\,\tau\,\texttt{of}\,b\,\texttt{in}\,m \parallel \mu \underset{\Sigma}{\longmapsto} \texttt{cls}\,\tau\,\texttt{of}\,b\,\texttt{in}\,m' \parallel \mu'}$$

$$\frac{e\ \mathsf{val}_{\Sigma} \quad b \notin e}{\texttt{cls}\,\tau\,\texttt{of}\,b\,\texttt{in}\,\texttt{ret}(e) \parallel \mu \underset{\Sigma, b\sim\tau}{\longmapsto} \texttt{ret}(e) \parallel \mu}$$

$$\frac{e\ \mathsf{val}_{\Sigma} \quad b \notin e}{\texttt{cls}\,\tau\,\texttt{of}\,b\,\texttt{in}\,\texttt{raise}(e) \parallel \mu \underset{\Sigma, b\sim\tau}{\longmapsto} \texttt{raise}(e) \parallel \mu}$$

(b) (8 points) Give the scoped dynamics for the raise command, following the pattern of the return command. Your solution will require *three* rules, two defining the execution and completion of the command itself, and one defining its interaction with the bind/otherwise (`bndow`) command.

Solution:

$$\frac{e\ \mathsf{val}_{\Sigma}}{\texttt{raise}(e)\ \mathsf{final}_{\Sigma}}$$

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\texttt{raise}(e) \parallel \mu \underset{\Sigma}{\longmapsto} \texttt{raise}(e') \parallel \mu}$$

$$\frac{e\ \mathsf{val}_{\Sigma}}{\texttt{bndow}\,x_1 \hookrightarrow \texttt{cmd}(\texttt{raise}(e))\,\texttt{in}\,m_1\,\texttt{ow}\,x_2 \hookrightarrow m_2 \parallel \mu \underset{\Sigma}{\longmapsto} [e/x_2]m_2 \parallel \mu}$$

(Question continues on the next page)

(c) (4 points) Give *two* examples of a class declaration command that that cannot *safely* make progress when executed, one involving a return command and one involving a raise command. Each example should take exactly *one* line, and should either be unable to progress, or would progress to an ill-defined state.

> **Solution:**
>
> 1. $\mathtt{cls}\, b\, \mathtt{of}\, \mathtt{nat}\, \mathtt{in}\, \mathtt{ret}(b \cdot \mathtt{z})$.
> 2. $\mathtt{cls}\, b\, \mathtt{of}\, \mathtt{nat}\, \mathtt{in}\, \mathtt{raise}(b \cdot \mathtt{z})$.

(d) (3 points) Define the statics of the raise command so as to ensure type safety.

> **Solution:** *Unfortunately, we determined after the exam that there is no way to give a type to the raise command that ensures safety.*
>
> The intended answer,
>
> $$\frac{\Gamma \vdash_\Sigma e : \mathtt{clsfd}}{\Gamma \vdash_\Sigma \mathtt{raise}(e) \mathrel{\dot\sim} \tau'}$$
>
> fails to ensure safety, even given the restrictions on class declarations in the next question, in the case of the command
>
> $$\mathtt{cls}\, b\, \mathtt{of}\, \mathtt{unit}\, \mathtt{in}\, \mathtt{raise}(b \cdot \langle\rangle)$$
>
> which transitions to $\mathtt{raise}(b \cdot \langle\rangle)$, with b thereby escaping its scope.
>
> *The correct conclusion is that one cannot consider scoped class declarations when exceptions raise values of type* `clsfd`*.*

(e) (3 points) Define the statics of class declaration so as to ensure type safety. In particular, the two counterexamples you gave should *not* be statically correct according to your rules!

> **Solution:**
>
> $$\frac{\Gamma \vdash_{\Sigma, b \sim \tau} m \mathrel{\dot\sim} \tau' \quad \tau' \ \mathsf{mobile} \quad \tau \ \mathsf{mobile}}{\Gamma \vdash_\Sigma \mathtt{cls}\{\tau\}(b.m) \mathrel{\dot\sim} \tau'}$$

(f) (4 points) Assuming that the class declaration and assignable declaration commands are the only means by which new symbols are introduced, is it sound to permit the type `clsfd` to be mobile, or must it be declared immobile?

> **Solution:** It cannot be treated as mobile. For example, one could then return a classified value from the scope of its declaration, or assign it to an assignable to achieve the same effect.

Question 5 [35]: Space Semantics

In class, we introduced cost dynamics to characterize the time needed to evaluate an expression. In this exercise, you will develop an analogous idea to study the *stack space* required during evaluation. Our case study will be **PCF**. We will work with both its evaluation semantics and its stack semantics. The various entities we need have the following (concrete) syntax:

$$\begin{array}{llll}
\textsf{type} & \tau & ::= & \texttt{nat} \\
& & & \tau_1 \rightharpoonup \tau_2 \\
\textsf{exp} & e & ::= & x \\
& & & \lambda\,(x:\tau)\,e \\
& & & e_1(e_2) \\
& & & \texttt{z} \\
& & & \texttt{s}(e) \\
& & & \texttt{ifz}(e; e_0; x.e_1) \\
& & & \texttt{fix}\,\tau : x\,\texttt{is}\,e
\end{array}
\qquad
\begin{array}{llll}
\textsf{frame} & f & ::= & \texttt{s}(-) \\
& & & \texttt{ifz}(-; e_0; x.e_1) \\
& & & \texttt{ap}(-; e_2) \\
& & & \texttt{ap}(e_1; -) \\
\textsf{stack} & k & ::= & \epsilon \\
& & & f; k
\end{array}$$

You may assume the usual static semantics and that all expressions in this exercise are well-typed. The rules for the stack machine for the call-by-value version of **PCF** are defined as follows.

$$\frac{}{k \rhd \texttt{z} \mapsto k \lhd \texttt{z}}\ (\texttt{rs:z}) \qquad \frac{}{k \rhd \texttt{s}(e) \mapsto \texttt{s}(-); k \rhd e}\ (\texttt{rs:s}) \qquad \frac{}{\texttt{s}(-); k \lhd v \mapsto k \lhd \texttt{s}(v)}\ (\texttt{ls:s})$$

$$\frac{}{k \rhd \texttt{ifz}(e; e_0; x.e_1) \mapsto \texttt{ifz}(-; e_0; x.e_1); k \rhd e}\ (\texttt{rs:ifz})$$

$$\frac{}{\texttt{ifz}(-; e_0; x.e_1); k \lhd \texttt{z} \mapsto k \rhd e_0}\ (\texttt{ls:ifz}_\texttt{z}) \qquad \frac{}{\texttt{ifz}(-; e_0; x.e_1); k \lhd \texttt{s}(v) \mapsto k \rhd [v/x]e_1}\ (\texttt{ls:ifz}_\texttt{s})$$

$$\frac{}{k \rhd \lambda\,(x:\tau)\,e \mapsto k \lhd \lambda\,(x:\tau)\,e}\ (\texttt{rs:}\lambda) \qquad \frac{}{k \rhd e_1(e_2) \mapsto \texttt{ap}(-; e_2); k \rhd e_1}\ (\texttt{rs:app})$$

$$\frac{}{\texttt{ap}(-; e_2); k \lhd v_1 \mapsto \texttt{ap}(v_1; -); k \rhd e_2}\ (\texttt{ls:app}_1)$$

$$\frac{}{\texttt{ap}(\lambda\,(x:\tau)\,e; -); k \lhd v_2 \mapsto k \rhd [v_2/x]e}\ (\texttt{ls:app}_2)$$

$$\frac{}{k \rhd \texttt{fix}\,x:\tau\,\texttt{is}\,e \mapsto k \rhd [\texttt{fix}\,x:\tau\,\texttt{is}\,e/x]e}\ (\texttt{rs:fix})$$

We define the *stack size* $|s|$ of a state s of the form $k \lhd e$ or $k \rhd v$ as the number $|k|$ of frames in the stack k. Given a sequence $s_0 \mapsto^* s_n$ of state transitions of the form $s_0 \mapsto s_1 \mapsto \ldots \mapsto s_{n-1} \mapsto s_n$, we define the *stack space* of this transition, written $\|s_0 \mapsto^* s_n\|$, as the size of the largest stack size in this sequence (i.e., the high-water mark). More precisely,

$$\|s_0 \mapsto s_1 \mapsto \ldots \mapsto s_{n-1} \mapsto s_n\| \quad = \quad \max_{i=0..n} \{|s_i|\}$$

(Question continues on the next page)

(a) (4 points) Describe the stack space that is used for evaluating the numeral $\overline{n}$ as a function of n, that is, define the function $\sigma_1(n) = \|\epsilon \rhd \overline{n} \mapsto^* \epsilon \lhd \overline{n}\|$ using a simple arithmetic expression.

Solution:
$\sigma_1(n) = n$

(b) (8 points) Now consider the usual definition of addition in **PCF**:

$$add \triangleq \texttt{fix}\, plus : \texttt{nat} \rightharpoonup \texttt{nat} \rightharpoonup \texttt{nat}\, \texttt{is}\, \lambda\,(x:\texttt{nat})\,\lambda\,(y:\texttt{nat})\,\texttt{ifz}(x;y;x'.\texttt{s}(plus(x')(y))),$$

Describe the stack space that is used for evaluating the expression $add(\overline{m})(\overline{n})$ as a function of m and n, that is, define the function $\sigma_2(n,m) = \|\epsilon \rhd add(\overline{m})(\overline{n}) \mapsto^* \epsilon \lhd \overline{m+n}\|$ using a simple arithmetic expression.

Hint: Write down the steps of the stack machine until you see a pattern. Use your answer to the previous questions.

Hint: We are asking for an exact solution but you can earn partial points with an asymptotically correct solution (using big-O).

Solution:
$\sigma_2(n,m) = \max(n+2, n+m+1)$

(c) (8 points) Next you will define the rules of a cost dynamics for stack space that correctly reflects the (high-water mark) stack space that we defined for the stack machine. The rules are based on the standard evaluation dynamics for **PCF** annotated with natural numbers. The resulting judgement has the form

$$e \Downarrow^s v$$

and intuitively means that expression e evaluates to value v using stack space s. The rules for conditionals and fix are defined as follows.

$$\frac{e \Downarrow^s \texttt{z} \qquad e_0 \Downarrow^{s_0} v_0}{\texttt{ifz}(e;e_0;x.e_1) \Downarrow^{\max\{s+1,s_0\}} v_0}\;(\texttt{ev:ifz}_\texttt{z})$$

$$\frac{e \Downarrow^s \texttt{s}(v) \qquad [v/x]e_1 \Downarrow^{s_1} v_1}{\texttt{ifz}(e;e_0;x.e_1) \Downarrow^{\max\{s+1,s_1\}} v_1}\;(\texttt{ev:ifz}_\texttt{s})$$

$$\frac{[\texttt{fix}\,x:\tau\,\texttt{is}\,e/x]e \Downarrow^s v}{\texttt{fix}\,x:\tau\,\texttt{is}\,e \Downarrow^s v}\;(\texttt{ev:fix})$$

(Question continues on the next page)

Define the remaining rules of the cost dynamics. The missing cases are zero, successor, abstraction, and applications.

Solution:

$$\frac{}{\texttt{z} \Downarrow^0 \texttt{z}}\ (\texttt{ev:z}) \qquad \frac{e \Downarrow^s v}{\texttt{s}(e) \Downarrow^{s+1} \texttt{s}(v)}\ (\texttt{ev:s})$$

$$\frac{}{\lambda\,(x:\tau)\,e \Downarrow^0 \lambda\,(x:\tau)\,e}\ (\texttt{ev:}\lambda) \qquad \frac{e_1 \Downarrow^{s_1} \lambda\,(x:\tau)\,e \quad e_2 \Downarrow^{s_2} v_2 \quad [v_2/x]e \Downarrow^s v}{e_1(e_2) \Downarrow^{\max\{s_1+1,s_2+1,s\}} v}\ (\texttt{ev:app})$$

(d) Consider the expression ω that is defined as follows.

$$\omega \triangleq \texttt{fix}\,\mathit{loop} : \texttt{nat}\,\texttt{is}\,\mathit{loop}$$

i. (3 points) Describe the stack space that is used for evaluating the expression ω on an empty stack ϵ.

Solution: The stack space used in the evaluation is 0 since $\|\epsilon \rhd \omega \mapsto^* \epsilon \rhd \omega\| = 0$.

ii. (3 points) What does the cost dynamics state about the stack space usage of ω?

Solution: With the cost dynamics we cannot derive a judgement for ω. Therefore it does not state anything about the stack usage.

(Question continues on the next page)

(e) (9 points) We now show that the cost dynamics is sound and complete with respect to the stack semantics: For any expression e and value v, it is the case that $\epsilon \rhd e \mapsto^* \epsilon \lhd v$ iff $e \Downarrow^s v$ where $s = \|\epsilon \rhd e \mapsto^* \epsilon \lhd v\|$.

Here, you will only prove the soundness of the evaluation dynamics. We also need to generalize the theorem to account for intermediate steps in the derivation. So the actual property you will prove is the following:

Property 1. *If $e \Downarrow^s v$ then for any k we have that $k \rhd e \mapsto^* k \lhd v$ and $\|k \rhd e \mapsto^* k \lhd v\| = s + |k|$.*

The proof proceeds by rule induction on $e \Downarrow^s v$. We will develop one case in detail and ask you to prove one more.

Case $\mathsf{ev{:}ifz_z}$**:** $\dfrac{e \Downarrow^s \mathtt{z} \qquad e_0 \Downarrow^{s_0} v_0}{\mathtt{ifz}(e; e_0; x.e_1) \Downarrow^{\max\{s+1, s_0\}} v_0}$ $(\mathtt{ev{:}ifz_z})$.

Induction hypothesis 1: For any stack k' there is a derivation of $k' \rhd e \mapsto^* k' \lhd \mathtt{z}$ such that $\|k' \rhd e \mapsto^* k' \lhd \mathtt{z}\| = s + |k'|$.

Induction hypothesis 2: For any stack k'' there is a derivation of $k'' \rhd e_0 \mapsto^* k'' \lhd v_0$ such that $\|k'' \rhd e_0 \mapsto^* k'' \lhd v_0\| = s_0 + |k''|$.

To show: For any stack k, there is a derivation of $k \rhd \mathtt{ifz}(e; e_0; x.e_1) \mapsto^* k \lhd v_0$ such that $\|k \rhd \mathtt{ifz}(e; e_0; x.e_1) \mapsto^* k \lhd v_0\| = \max\{s+1, s_0\} + |k|$.

Proof:
Assume k is given. Let k' be $\mathtt{ifz}(-; e_0; x.e_1); k$ and k'' be k. Then $|k'| = |k| + 1$ and $|k''| = |k|$ and, using IH 1 with k' and IH 2 with k'', we derive

$$k \rhd \mathtt{ifz}(e; e_0; x.e_1) \mapsto \mathtt{ifz}(-; e_0; x.e_1); k \rhd e \mapsto^* \mathtt{ifz}(-; e_0; x.e_1); k \lhd \mathtt{z} \mapsto k \rhd e_0 \mapsto^* k \lhd v_0$$

Moreover, we have

$$\begin{array}{rl}
& \left\| \begin{array}{ll} & k \rhd \mathtt{ifz}(e; e_0; x.e_1) \\ \mapsto & \mathtt{ifz}(-; e_0; x.e_1); k \rhd e \\ \mapsto^* & \mathtt{ifz}(-; e_0; x.e_1); k \lhd \mathtt{z} \\ \mapsto & k \rhd e_0 \\ \mapsto^* & k \lhd v_0 \end{array} \right\| \\
= & \max \left\{ \begin{array}{l} |k \rhd \mathtt{ifz}(e; e_0; x.e_1)|, \\ \|\mathtt{ifz}(-; e_0; x.e_1); k \rhd e \mapsto^* \mathtt{ifz}(-; e_0; x.e_1); k \lhd \mathtt{z}\|, \\ \|k \rhd e_0 \mapsto^* k \lhd v_0\| \end{array} \right\} \\
= & \max\{|k|, s + 1 + |k|, s_0 + |k|\} \qquad \text{(by IH 1 and IH 2)} \\
= & \max\{s+1, s_0\} + |k|
\end{array}$$

This completes the proof for the case $\mathsf{ev{:}ifz_z}$.

(Question continues on the next page)

Provide the induction proof for Property 1 in the case where the derivation of $e \Downarrow^s v$ ends in rule ev:s.

Case ev:s: $\dfrac{e \Downarrow^s v}{\mathtt{s}(e) \Downarrow^{s+1} \mathtt{s}(v)}$ (ev:s).

Induction hypothesis:

To show:

Proof:

Solution:

Case ev:s: $\dfrac{e \Downarrow^s v}{\mathtt{s}(e) \Downarrow^{s+1} \mathtt{s}(v)}$ (ev:s).

Induction hypothesis: For any stack k' there is a derivation of $k' \triangleright e \mapsto^* k' \triangleleft v$ such that $\|k' \triangleright e \mapsto^* k' \triangleleft v\| = s + |k'|$.

To show: For any stack k, there is a derivation of $k \triangleright \mathtt{s}(e) \mapsto^* k \triangleleft \mathtt{s}(v)$ such that $\|k \triangleright \mathtt{s}(e) \mapsto^* k \triangleleft \mathtt{s}(v)\| = s + 1 + |k|$.

Proof: Assume k is given. Let k' be $\mathtt{s}(-); k$. Then, using the IH for k',

$$k \triangleright \mathtt{s}(e) \mapsto \mathtt{s}(-); k \triangleright e \mapsto^* \mathtt{s}(-); k \triangleleft v \mapsto k \triangleleft \mathtt{s}(v)$$

is a derivation of $k \triangleright \mathtt{s}(e) \mapsto^* k \triangleleft \mathtt{s}(v)$. Moreover,

$$\begin{array}{rllll} & \|k \triangleright \mathtt{s}(e) \mapsto & \mathtt{s}(-); k \triangleright e \mapsto^* \mathtt{s}(-); k \triangleleft v & \mapsto k \triangleleft \mathtt{s}(v)\| & \\ = & \max\{ \quad |k|, & \|\mathtt{s}(-); k \triangleright e \mapsto^* \mathtt{s}(-); k \triangleleft v\|, & |k| \quad \} & \textbf{(by IH)} \\ = & \max\{ \quad |k|, & t + 1 + |k|, & |k| \quad \} & \\ = & t + 1 + |k| & & & \end{array}$$