There are 15 pages in this examination, comprising 6 questions worth a total of 175 points.

You may refer to your personal notes, course notes and supplements, and to Practical Foundations for Programming Languages, but not to any other person or source.

You may use your laptop or tablet as long as you only refer to the aforementioned sources and disable WiFi and other network connections at all times.

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.

All figures mentioned in the exam appear in the appendix so that they may be easily torn off for reference.

<table>
<thead>
<tr>
<th>Question:</th>
<th>Short</th>
<th>Safety</th>
<th>Cost</th>
<th>Cont’s</th>
<th>Rep Ind</th>
<th>Rec. Types</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points:</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>25</td>
<td>175</td>
</tr>
<tr>
<td>Score:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Question 1 [30]: Short Answers
about 5-10 short answer questions: true/false, fill-in one line.

(a) [3 points] Define a System F value that has the following type:
\[ \forall t. (\forall s. s \rightarrow t) \rightarrow t \]

**Solution:** \[ \Lambda t. \lambda (f : \forall s. s \rightarrow t) f[t \rightarrow t](\lambda(x : t)x) \]

(b) [3 points] Define an expression in FPC that loops forever. Do not use fix/self/unroll.

**Solution:** \( (\lambda(x : \text{rec}(t.t \rightarrow t)) \text{unfold}(x)(x))(\text{fold}(\lambda(x : \text{rec}(t.t \rightarrow t)) \text{unfold}(x)(x))) \)

(c) [3 points] In Modernized Algol, the judgement \( \tau \text{ mobile} \) defines which types can be returned from a dcl command. If we extend the definition of \( \tau \text{ mobile} \) with the rule
\[ \tau_1 \text{ mobile} \quad \tau_2 \text{ mobile} \]
\[ \frac{}{\tau_1 \rightarrow \tau_2 \text{ mobile}} \]

the language becomes unsafe. Give an expression in Modernized Algol with this definition of \( \tau \text{ mobile} \) that violates type safety.

**Solution:** \[ \text{dcl}(a. \text{ret}(\lambda(x : \text{nat}) (\lambda(y : \text{nat cmd}) x)(\text{cmd}(\text{get}[a])))) \]

(d) Consider the following theorem in first-order logic, where \( P \) is a predicate that refers to \( x \).
\[ \forall x. P(x) \implies \exists x. P(x) \]

i. [1 point] Express the statement as a type in System FE.

**Solution:** \( \forall(x.P) \rightarrow \exists(x.P) \)

ii. [3 points] Prove the statement by providing an instance of the type you wrote.

**Solution:** \( \lambda(y : \forall(x.P)) \text{ pack } \forall(t.t) \text{ with } y[\forall(t.t)] \text{ as } \exists(x.P) \)

(e) [3 points] Consider the following language: PCF by-name, as given in lecture, with the single change that instead of having the general fix, we instead have the construct fun \( f \ (x : \tau) : \tau' \text{ is e} \), which defines recursive functions, and which is a value. True or false: every value of type nat in this language necessarily converges to a value in System PCF by-value. If true, provide a brief explanation why, or a counterexample if it is false.

**Solution:** False. \( s((\text{fun } f \ (x : \text{nat}) : \text{nat} \text{ is } f(x)) \text{ z}) \) does not converge.

(f) [3 points] True or false: because of System F’s support for universal types, any function on the natural numbers in DPCF may be translated to an equivalent function in System F. Give a brief explanation for your answer.
Solution: False. System F cannot express nonterminating computations, unlike DPCF.

(g) [3 points] Consider the standard Church encoding of natural numbers in System F:

\[
\begin{align*}
nat & \triangleq \forall(t.t \to (t \to t) \to t) \\
zero & \triangleq \Lambda t. \lambda(z : t). \lambda(s : t \to t). z \\
succ & \triangleq \lambda(n : \forall(t.t \to (t \to t) \to t)). \Lambda t. \lambda(s : t \to t). s(n(t)(z)(s))
\end{align*}
\]

Implement addition using this Church encoding of natural numbers in System F.

Solution:

\[
\begin{align*}
\lambda(n : \forall(t.t \to (t \to t) \to t)) \lambda(m : \forall(t.t \to (t \to t) \to t)) \ m[n(t \to t)](n)(\text{succ})
\end{align*}
\]

(h) [4 points] As a reminder, the static semantics for continuations is given by the following rules:

\[
\begin{align*}
\Gamma, x : \tau \cont \vdash e : \tau \\
\Gamma \vdash \text{letcc}\{\tau\}(x.e) : \tau \\
\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_1 \cont \\
\Gamma \vdash \text{throw}\{\tau\}(e_1; e_2) : \tau
\end{align*}
\]

Write a term of type \(\cont(\cont(\tau_1) + \cont(\tau_2)) \to (\tau_1 \times \tau_2)\).

Solution:

\[
\begin{align*}
\tau & \triangleq \cont(\tau_1) + \cont(\tau_2) \\
e_1 & \triangleq \text{letcc}\{\tau_1\}(y. \text{throw}\{\tau_1\}(1 : y;x)) \\
e_2 & \triangleq \text{letcc}\{\tau_2\}(z. \text{throw}\{\tau_2\}(r : z;x)) \\
\lambda(x : \cont(\tau))\langle e_1, e_2 \rangle
\end{align*}
\]

(i) [4 points] Consider three channels \(a, b, c\) such that \(a \sim \tau_1 + \tau_2\), \(b \sim \tau_1\), and \(c \sim \tau_2\). Using the definition of Concurrent Algol from Assignment 6, define an expression \(e_{\text{forwarder}}\) such that \(\text{spawn}(e_{\text{forwarder}})\) will create a process that loops forever while forwarding messages from channel \(a\) to channel \(b\) if the message has type \(\tau_1\) and to channel \(c\) if the message has type \(\tau_2\).

Solution:

\[
\begin{align*}
\text{fix forwarder} : \text{cmd}(\text{unit}) \text{ is cmd}\{ \\
\quad \text{bnd(cmd(ret(rcv[a]))), msg.} \\
\quad \text{bnd(case msg of} \\
\qquad L . x -> \text{cmd(emit[b](x))} \\
\qquad R . y -> \text{cmd(emit[c](y))} \\
\qquad _ . \text{bnd(forwarder; z.z))}
\}
\end{align*}
\]
Question 2 [30]: Type Safety
Consider a dynamics for MA using a stack machine that represents both control information (the context of execution) and data information (the active assignables). Doing so corresponds to a conventional “run-time stack” in the sense of C or similar languages. In this dynamics, stacks are defined by the following grammar:

\[ k ::= \epsilon \mid k ; \text{bnd}(\text{cmd}(\_); x.e) \mid k ; \text{dcl}\{\tau_1\}\{e_1; a.\_\_\} \]

The second form of stack frame represents a sequential composition in which a suspended command is currently being executed. The third form of frame represents the declaration of an assignable with contents \( e_1 \), which will be constrained to be a value. These are the only frames because pure expression evaluation remains as defined in PFPL, with transitions \( e \mapsto e' \) and values \( e \text{ val}_\Sigma \). Execution states for commands have the form \( k \#_\Sigma \Sigma \), the juxtaposition of a command with a stack, with \( \Sigma \) governing the active assignables. The rules defining this abstract machine, but for the rules governing get and set, are given in Figure 1. It makes use of two auxiliary judgments, \( k @ a = e \) which returns the value associated to \( a \) by a unique declaration in \( k \), and \( k @ a \leftarrow e \mapsto k' \), which updates the contents of \( a \) in \( k \) to obtain \( k' \). These are not defined here.

Define the judgement \( s \text{ ok} \) for \( s \) a state of the abstract machine by the following rule:

\[
\frac{\text{ok} \vdash \Sigma k \div \tau \quad \text{ok} \vdash \Sigma m \sim \tau}{k \#_\Sigma m \text{ ok}}
\]

That is, a state is well-formed if the stack is well-formed and expects a return value of the type returned by the command. Stack typing is defined by the following three rules:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Precondition</th>
<th>Postcondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPTY</td>
<td>( \vdash_\Sigma \epsilon \div \tau )</td>
<td>( k #_\Sigma m \text{ ok} )</td>
</tr>
<tr>
<td>BIND</td>
<td>( \vdash_\Sigma k \div \tau_2 ) \quad x_1 : \tau_1 \vdash_\Sigma m_2 \sim \tau_2 )</td>
<td>( k #_\Sigma m \text{ ok} )</td>
</tr>
<tr>
<td>DCL</td>
<td>( \vdash_\Sigma k \div \tau_2 ) \quad \vdash_\Sigma e_1 : \tau_1 ) \quad \vdash_{\Sigma,a \sim \tau_1} k #_\Sigma \text{ dcl}{\tau_1}{e_1; a.__} \div \tau_2 )</td>
<td></td>
</tr>
</tbody>
</table>

The goal of this exercise is to prove type safety of the statics for MA given in PFPL relative to the stack machine dynamics given in Figure 1. As usual, the proof of safety breaks into progress and preservation. You are to prove several cases of the preservation theorem as directed below.

Prove by induction on the stack machine dynamics the following cases of the preservation theorem: if \( k \#_\Sigma m \mapsto k' \#_\Sigma m' \), then \( k \#_\Sigma m \text{ ok} \) implies \( k \#_\Sigma m' \text{ ok} \).

State, but do not prove, such substitution lemmas for typing as you may require for your proof.

State, but do not prove, such lemmas for stack, command, or expression typing to limit the assumed declarations as you may require for your proof.

Hint: a correct proof will require at least one of each of these lemmas!

(a) [7 points] Rule BIND-CMD:

**Solution:** By assumption \( \vdash_\Sigma k \div \tau_2 \) and \( \vdash_\Sigma \text{bnd}(\text{cmd}(m_1); x.m_2) \sim \tau_2 \). By three applications of inversion \( m_1 \sim \tau_1 \), and \( x : \tau_1 \vdash_\Sigma m_2 \sim \tau_2 \).

But then \( \vdash_\Sigma k ; \text{bnd}(\text{cmd}(\_); x.m_2) \div \tau_1 \) and \( \vdash_\Sigma m_1 \sim \tau_1 \), which is enough for the result.
(b) [7 points] Rule BIND-RET:

**Solution:** By assumption $\vdash_{\Sigma} k ; \text{bind}(\text{cmd}(-); x.m_2) \div \tau_1$ and $\vdash_{\Sigma} \text{ret}(e_1) \sim \tau_1$. By several uses of inversion $\vdash_{\Sigma} k \div \tau_2$, $x : \tau_1 \vdash_{\Sigma} m_2 \sim \tau_2$, and $\vdash_{\Sigma} e_1 : \tau_1$. Moreover, by the premise of the rule, $e_1 \text{ val}_\Sigma$. By substitution of values for variables in commands $\vdash_{\Sigma} [e_1/x]m_2 \sim \tau_2$, which is enough for the result.

(c) [7 points] Rule DCL-BODY.

**Solution:** By assumption $\vdash_{\Sigma} k \div \tau_2$ and $\vdash_{\Sigma} \text{dcl}\{\tau_1\}(e_1;a.m_2) \sim \tau_2$. Moreover, $e_1 \text{ val}_\Sigma$. By inversion $\vdash_{\Sigma} e_1 : \tau_1$, $\vdash_{\Sigma,a \sim \tau_1} m_2 \sim \tau_2$. Consequently, $\vdash_{\Sigma,a \sim \tau_1} k ; \text{dcl}\{\tau_1\}(e_1;a.m_2) \div \tau_2$, which suffices for the result.

(d) [9 points] Rule DCL-RET.

**Solution:** By assumption $\vdash_{\Sigma,a \sim \tau_1} k ; \text{dcl}\{\tau_1\}(e_1;a.\sim) \div \tau_2$, $\vdash_{\Sigma,a \sim \tau_1} \text{ret}(e_2) \sim \tau_2$. By inversion $\vdash_{\Sigma,a \sim \tau_1} k \div \tau_2$ and $\vdash_{\Sigma,a \sim \tau_1} e_2 : \tau_2$. By mobility of both $\tau_1$ and $\tau_2$, these judgments may be strengthened to $\vdash_{\Sigma} k \div \tau_2$ and $\vdash_{\Sigma} e_2 : \tau_2$. But these suffice for the result.
Question 3 [30]: Cost Dynamics for Parallelism with Exceptions

Recall the formulation of parallelism in PCF given in the supplement entitled “Types and Parallelism.” The parallel product type is there given the following statics:

\[
\Gamma \vdash e_1 \sim \cdot \cdot \cdot \tau_1 \\
\Gamma \vdash e_2 \sim \cdot \cdot \cdot \tau_2 \\
\Gamma \vdash e_1 & e_2 : \tau_1 & \tau_2
\]

PAR-BIND

\[
\Gamma \vdash v : \tau_1 & \tau_2 \\
x : \tau_1 \otimes \tau_2 \vdash e \sim \tau \\
\Gamma \vdash \text{par}(v; x.e) \sim \tau
\]

The idea is that the components of the product are evaluated, their values are formed into an eagerly evaluated pair, and passed to the joint point of the fork of the two computations.

Extend this language with exceptions, leaving the type `exn` of exception values unspecified. The possibility of exceptions is incorporated into the elimination form for the product type as indicated by the following typing rules:

RAISE

\[
\Gamma \vdash \Sigma v : \text{exn} \\
\Gamma \vdash \Sigma \text{raise}(v) \sim \tau
\]

PAR-TRY

\[
\Gamma \vdash \Sigma x : \tau_1 \otimes \tau_2 \vdash \Sigma e_2 \sim \tau \\
\Gamma, x : \text{exn} \vdash \Sigma e_3 \sim \tau \\
\Gamma \vdash \Sigma \text{par}(v_1; x.e_2; x.e_3) \sim \tau
\]

The elimination form for the product type is generalized to handle exceptions that may arise during the evaluation of the two encapsulated computations. The elimination form must evaluate both components of the pair before continuing with either the normal or the exceptional return. If neither raises an exception, the pair of values resulting from the computations is passed to the normal return point. If either or both raise an exception, the leftmost of these is passed to the exceptional return point.

(a) Give the transition dynamics for the generalized parallel pair elimination form that accounts for exceptions in the manner just described. Follow the description carefully!

i. [3 points] Assuming that the first argument is a value, give the rule for the parallel evaluation of the two encapsulated computations:

Solution:

\[
\text{PAR} \\
\begin{array}{c}
\quad e_1 \mapsto e_1' \\
\quad e_2 \mapsto e_2' \\
\quad \text{par}(e_1 & e_2; x.e_3; y.e_4) \mapsto \text{par}(e_1' & e_2'; x.e_3; y.e_4)
\end{array}
\]

ii. [6 points] Two flavors of ketchup are required. Give the catch-up rules pertaining to the exceptional cases.

Solution:

\[
\begin{array}{c}
\text{PAR-EXC-L} \\
\quad e_2 \mapsto e_2' \\
\quad \text{par}(\text{raise}(v_1) & e_2; x.e_3; y.e_4) \mapsto \text{par}(\text{raise}(v_1) & e_2'; x.e_3; y.e_4)
\end{array}
\]

\[
\begin{array}{c}
\text{PAR-EXC-R} \\
\quad e_1 \mapsto e_1' \\
\quad \text{par}(e_1 & \text{raise}(v_2); x.e_3; y.e_4) \mapsto \text{par}(e_1' & \text{raise}(v_2); x.e_3; y.e_4)
\end{array}
\]

iii. [9 points] Give the three rules pertaining to exception propagation:
(b) [12 points] Give the cost dynamics for the extended parallel pair elimination form that takes account of both “normal” and “exceptional” returns. Make use of the following two forms of judgment:

1. $e \downarrow^c v$: expression $e$ evaluates to value $v$ with cost $c$.
2. $e \uparrow^c v$: expression $e$ raises the exception value $v$ with cost $c$.

It is handy to make use of the combined $e \updownarrow^c v$, which means that $e$ evaluates to or raises the value $v$ with cost $c$. When used in a rule the sense is to be the same in the conclusion as in the premise (both evaluate or both raise). For example, here is the cost dynamics for the case that the principal argument and both sub-computations evaluate normally.

\[
\text{PAR-VAL-VAL}\quad \begin{array}{l}
 e_1 \downarrow^{c_1} v_1 \\
 e_2 \downarrow^{c_2} v_2
\end{array} \quad \begin{array}{c}
 [v_1 \otimes v_2 / x] e_3 \uparrow^{c_3} v_3
\end{array} \quad \Rightarrow \quad \begin{array}{c}
 \text{par}(e_1 \& e_2; x.e_3; y.e_4) \updownarrow ((c_1 \otimes c_2) \oplus c_3 \oplus 1) v_4
\end{array}
\]

Give the cost dynamics for the two cases corresponding to an exceptional outcome for either of the parallel sub-computations.

\[
\text{Solution:}
\]

\[
\begin{array}{c}
\text{PAR-VAL-EXC}\quad \begin{array}{l}
 e_1 \downarrow^{c_1} v_1 \\
 e_2 \uparrow^{c_2} v_2
\end{array} \quad \begin{array}{c}
 [v_2 / y] e_4 \uparrow^{c_4} v_4
\end{array} \quad \Rightarrow \quad \begin{array}{c}
 \text{par}(e_1 \& e_2; x.e_3; y.e_4) \updownarrow ((c_1 \otimes c_2) \oplus c_4 \oplus 1) v_4
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\text{PAR-EXC-EXC-VAL}\quad \begin{array}{l}
 e_1 \uparrow^{c_1} v_1 \\
 e_2 \uparrow^{c_2} v_2
\end{array} \quad \begin{array}{c}
 [v_1 / y] e_4 \downarrow^{c_4} v_4
\end{array} \quad \Rightarrow \quad \begin{array}{c}
 \text{par}(e_1 \& e_2; x.e_3; y.e_4) \downarrow ((c_1 \otimes c_2) \oplus c_4 \oplus 1) v_4
\end{array}
\end{array}
\]
Question 4 [30]: Programming with Continuations

This question explores the use of continuations to implement two different exception control mechanisms. As a warm-up, you are asked to give solutions to a few programming problems inspired by the interpretation of \( \text{cont}(\tau) \) as corresponding to negation in classical logic. You are then asked to use these ideas to implement exceptions using continuations.

(a) Define functions using continuations according to the following specifications:

i. [4 points] Define \( \text{dne} : \text{cont}(\text{cont}(\tau)) \rightarrow \tau \) such that

\[
\text{if } k \triangleright \text{ap}(\text{dne}; v) \mapsto \star k \triangleleft w, \text{ then } k \triangleright \text{throw}(\text{cont}(k); v) \mapsto \star k \triangleleft w.
\]

Solution:

\[
\text{dne} \triangleq \lambda (k:k) \text{letcc}(r.\text{throw}(r; kk)).
\]

ii. [6 points] Define \( \text{cp} : (\tau_1 \rightarrow \tau_2) \rightarrow \text{cont}(\tau_2) \rightarrow \text{cont}(\tau_1) \) such that if a value \( v_1 \) of type \( \tau_1 \) is thrown to \( \text{cp}(f)(k) \), then \( f(v_1) \) is thrown to \( k \):

Solution:

\[
\text{cp} \triangleq \lambda (f:\tau_1 \rightarrow \tau_2) \lambda (k_2:\text{cont}(\tau_2)) \text{letcc}(r.\text{throw}(f(\text{letcc}(k_1.\text{throw}(k_1;r)); k_2)); k_2)).
\]

iii. [4 points] Define \( \text{dm} : \text{cont}(\tau_1 + \tau_2) \rightarrow (\text{cont}(\tau_1) \times \text{cont}(\tau_2)) \) such that if \( v_1 : \tau_1 \) is thrown to the first component of the application \( \text{dm}(k) \), then \( 1 \cdot v_1 \) is thrown to \( k \), and analogously for the second component.

Solution:

\[
\text{dm} \triangleq \lambda (k:\text{cont}(\tau_1 + \tau_2)) \langle e_1, e_2 \rangle, \text{ where } e_1 \triangleq \text{letcc}(r_1.\text{throw}(1 \cdot \text{letcc}(k_1.\text{throw}(k_1;r_1)); k)), \text{ and } e_2 \triangleq \text{letcc}(r_2.\text{throw}(r \cdot \text{letcc}(k_2.\text{throw}(k_2;r_2)); k)).
\]

(b) The exception control mechanism described in Chapter 29 of PFPL may be implemented using continuations. The idea is that a \( \text{try} \) expression establishes a continuation that is used by a \( \text{raise} \) expression to transfer control to it. To do the implementation properly requires taking into account that \( \text{try} \)'s can be nested within one another, with the innermost one taking precedence over the outer. Moreover, any \( \text{raise} \) performed within the handler should be propagated to the surrounding handler, if any. The implementation should also account for uncaught exceptions using a “backstop” continuation to be described shortly.

To manage the nesting, assume given the following operations on an implicit ephemeral stack of continuations:

1. \( \text{reset} : \text{unit} \rightarrow \text{unit} \): erase the stack.
2. \( \text{push} : \text{exn cont} \rightarrow \text{unit} \): push a continuation onto the stack.
3. **pop**: unit → exn cont: removes the topmost element of the stack and returns it.

The **pop** operation assumes that the stack is non-empty (and aborts otherwise, with an uncaught exception error.)

Some hints for the solution:

1. It is easy to misimplement the handler in such a way that an infinite loop occurs if the handler itself raises an exception by causing the handler to re-invoke the handler. Take care to avoid this.

2. It is necessary to use the programming methods exercised by the warm-up examples, in particular the idea of throwing a value to the return point.

Feel free to write $e_1; e_2$ for let be $e_1$ in $e_2$, so that $e_1$ is executed solely for its effect, throwing away its value.

i. [6 points] Define the computation raise($e$) ∼ $\tau$ that raises the exception value $e : \text{exn}$ and passes it to the nearest enclosing handler:

**Solution:** 

\[
\text{raise}(e) \triangleq \text{throw } e \text{ to } \text{pop}(()).
\]

ii. [10 points] Define the bind operation $\text{bnd}(e; x.e_1; y.e_2) \sim \rho$, where $e : \text{comp}(\tau)$, $x : \tau \vdash e_1 \sim \rho$, and $y : \text{exn} \vdash e_2 \sim \rho$. It evaluates the encapsulated computation given by $e$, passing its return value to $e_1$ if it returns normally, and passing its raised value to $e_2$ if it raises an exception.

**Solution:**

\[
\text{bnd}(e; x.e_1; y.e_2) \triangleq \text{letcc ret in (I), where}
\]

\[
(I) \triangleq \text{let } y \text{ be (letcc hdlr in (push(hdlr);(II))) in } e_2
\]

\[
(II) \triangleq \text{let } x \text{ be } e \text{ in (pop(()); throw } e_1 \text{ to ret)}
\]
Question 5 [30]: Representation Independence of Abstract Types

Consider the following (minimal) abstract type of numbers:

\[ \tau \triangleq \exists t. (\text{zero} \mapsto t, \text{succ} \mapsto t, \text{plus} \mapsto t \times t \mapsto t, \text{isz} \mapsto t \mapsto \text{bool}) \]

This abstraction admits two implementations, one in which the numbers are represented in unary and one in binary. This existential type corresponds to the SML signature

signature NUM = sig
type t
val zero : t
val succ : t -> t
val plus : t * t -> t
val isz : t -> bool
end

Two SML implementations of NUM are given in Figures 2 and 3.

In this question you are to prove that these two implementations, notated I and II, are interchangeable by showing that each operation preserves the following simulation relation between the two representation types: for values \( u : \text{un} \) and \( b : \text{bin} \), and, define

\[ u \sim b \quad \text{iff} \quad ||u||^I = ||b||^II, \]

where \( ||u||^I \) is the numeric value of \( u \) as a unary numeral, and \( ||b||^II \) is the numeric value of \( b \) viewed as a binary numeral:

\[
\begin{align*}
||Z||^I &= 0 \\
||S(u)||^I &= ||u||^I + 1 \\
||Z0||^II &= 0 \\
||D0(b)||^II &= 2 \times ||b||^II \\
||D1(b)||^II &= 2 \times ||b||^II + 1
\end{align*}
\]

For concision, abbreviate \( ||u|| = ||u||^I \) and \( ||b|| = ||b||^II \).

(a) [2 points] Show that \( \text{zero}^I \sim \text{zero}^II \):

Solution:

\[ ||\text{zero}^I|| = ||Z|| = 0 = ||Z0|| = ||\text{zero}^II||. \]

(b) [6 points] Show, by induction on the structure of \( b \), that \( ||\text{succ}^II(b)|| = ||b|| + 1 \):

Solution:

1. \( b = Z0 \): \( ||\text{succ}^II(b)|| = ||\text{succ}^II(Z0)|| = ||D1(Z0)|| = 2 \times 0 + 1 = ||b|| + 1. \)
2. \( b = D0(b') \): \( ||\text{succ}^II(b)|| = ||\text{succ}^II(D0(b'))|| = ||D1(b')|| = 2 \times ||b'|| + 1 = ||b|| + 1. \)
3. \( b = D1(b') \): By induction \( ||\text{succ}^II(b')|| = ||b'|| + 1 \), and so

\[ ||\text{succ}^II(b)|| = ||\text{succ}^II(D1(b'))|| = ||D0(\text{succ}^II(b'))|| = 2 \times (||b'|| + 1) = ||b|| + 1. \]
(c) [2 points] Show that if \( u \sim b \), then \( succ^I(u) \sim succ^H(b) \):

**Solution:** We are given \( ||u|| = ||b|| \). Applying the previous, we have

\[
||succ^I(u)|| = ||S(u)|| = ||u|| + 1 = ||b|| + 1 = ||succ^H(b)||
\]

(d) [6 points] Show, by induction on the structure of \( b \), that if \( isz^H(b) = true \) if \( ||b|| = 0 \) and \( isz^H(b) = false \) otherwise:

**Solution:**

1. \( b = \text{Z0} \): immediate.

2. \( b = \text{D0}(b') \): Then \( isz^H(b) = isz^H(b') \) and \( ||b|| = ||b'|| \). If \( ||b|| = true \), then \( ||b'|| = 0 \) and by induction \( isz^H(b') = true \), completing the case. Otherwise, \( ||b'|| \neq 0 \) and by induction \( isz^H(b') = false \), as desired.

3. \( b = \text{D1}(b') \): Then \( ||b|| \neq 0 \) and \( isz^H(b) = false \).

(e) [9 points] Show that if \( u \sim b \), then \( isz^I(u) = isz^H(b) \):

**Solution:** We are given \( ||u|| = ||b|| \). Proceed by induction on the structure of \( b \), using the previous.

1. \( b = \text{Z0} \): Because \( ||b|| = 0 = ||u|| \), it follows that \( u = Z \), and hence \( isz^I(u) = true = isz^H(b) \).

2. \( b = \text{D0}(b') \): We have \( ||b|| = 2 \times ||b'|| = ||u|| \) and \( ||isz^H(b)|| = ||isz^H(b')|| \). Either \( ||b'|| = 0 \) or not. If so, then \( ||u|| = 0 \), so \( u = Z \) and \( isz^I(u) = true \). But then by the previous \( isz^H(b') = true \) and so \( isz^H(b) = true \). If not, then \( ||u|| \neq 0 \) so \( u = S(u') \) for some \( u' \), and \( isz^I(u) = false = isz^H(b) \), again by the previous.

3. \( b = \text{D1}(b') \): Then \( isz^H(b) = false \), and \( ||b|| = 2 \times ||b'|| + 1 \), so by assumption \( ||u|| \neq 0 \) and so \( u = S(u') \) and \( isz^I(u) = false \) as well.

(f) [5 points] Show that if \( u \sim b \) and \( v \sim c \), then \( plus^I(u,v) \sim plus^H(b,c) \). Proceed by simultaneous induction on the structure of \( b \) and \( c \). Consider only the case \( b = \text{D1}(b') \) and \( c = \text{D1}(c') \).

**Solution:** In that case \( ||b|| = 2 \times ||b'|| + 1 \), \( ||c|| = 2 \times ||c'|| + 1 \), and \( plus^H(b,c) = \text{D0}(succ^H(plus^H(b',c'))) \). By the assumptions \( ||b|| = ||u|| \neq 0 \) and \( ||c|| = ||v|| \neq 0 \), so \( u = S(u') \), \( v = S(v') \), and \( plus^I(u,v) = S(S(plus^I(u',v'))) \). Thus \( ||u|| = ||u'|| + 1 \), \( ||v|| = ||v'|| + 1 \), and so \( ||u'|| = 2 \times ||b'|| \) and \( ||v'|| = 2 \times ||c'|| \). Calculating

\[
||plus^H(b,c)|| = 2 \times ||b'|| + 2 \times ||c'|| + 2 = ||u'|| + ||v'|| + 2 = ||plus^I(u,v)||.
\]
Question 6 [25]: Inductive, Coinductive, and Recursive Types

Consider two variants of FPC, a by-name form in which pairing, injections, and recursive foldings are lazy and functions are called by name, and a by-value form in which pairing, injection, and recursive folding are eager, and functions are called by value.

Using recursive types we may define, in the by-name variant, a general fixed point operation \( \text{fix}\{\tau\}(x.e) \) and, in the by-value variant, a self-referential function, \( \text{fun}\{\tau_1;\tau_2\}(f.x.e) \). You may use either of these in the relevant situations (by-name or by-value).

Inductive types, which are characterized by the recursive foldings, are a natural fit for the by-value variant of FPC, whereas co-inductive types, which are characterized by the recursive unfoldings, are a natural fit for the by-name variant.

For example, \( \text{tree} \) is an inductive type whose elements are built up from the empty tree by forming binary nodes. It may be defined in FPC-by-value as follows:

\[
\text{tree} \triangleq \text{rec}(t.1 + (t \times t)) \\
\text{empty} \triangleq \text{fold}(1 \cdot \langle\rangle) \\
\text{node}(t_1,t_2) \triangleq \text{fold}(r \cdot (t_1,t_2))
\]

Under an eager interpretation the values of type \( \text{tree} \) are just the bare binary trees built up from \( \text{empty} \) by applications of \( \text{node} \).

Dually, \( \text{itree} \) is a coinductive type from whose elements one may compute its two immediate subtrees, if it has any. It may be defined in FPC-by-name as follows:

\[
\text{itree} \triangleq \text{rec}(t.1 + (t \times t)) \\
\text{subtrees}(t) \triangleq \text{unfold}(t)
\]

Thus, if \( t \) is of type \( \text{itree} \), then \( \text{subtrees}(t) \) is of type \( 1 + (\text{itree} \times \text{itree}) \), revealing whether the infinite tree has any immediate subtrees, and, if so, the pair of infinite trees that are its two immediate subtrees.

For the type operator \( t.1 + (t \times t) \), the expression \( \text{treemap}(x.e';e) \) is the generic extension \( \text{map}\{t.\tau\}(x.e')(e) \), which instantiates \( x.e' \) at every position in \( e \) corresponding to the occurrences of \( t \) in \( 1 + (t \times t) \). Here is a specialized typing rule for this instance of generic extension:

\[
\frac{}{\Gamma \vdash \Delta e : \tau \quad \Gamma, x : \sigma \vdash e' : \sigma' \quad \Gamma \vdash e : 1 + (\sigma \times \sigma) \quad \Gamma \vdash \Delta \text{treemap}(x.e';e) : 1 + (\sigma' \times \sigma')}{\Gamma \vdash \Delta \text{treemap}(x.e';e) : 1 + (\sigma' \times \sigma')}
\]

(a) [5 points] Using \( \text{treemap} \) define a function of type \( 1 + (\text{nat} \times \text{nat}) \rightarrow (1 + (\text{bool} \times \text{bool})) \) that tests each subtree position for the number zero.

Solution:

\[
\lambda(x : 1 + (\text{nat} \times \text{nat}))\text{treemap}(n.\text{ifz } n \{\text{z } \rightarrow \text{true } | \text{s(z) } \rightarrow \text{false}\};x).
\]

(b) [10 points] In FPC-by-value specify and define the recursor for the inductive type \( \text{tree} \) represented as above. It should satisfy the following typing rule:

\[
\frac{\Gamma \vdash e : \text{tree} \quad \Gamma, x : 1 + (\rho \times \rho) \vdash e' : \rho}{\Gamma \vdash \Delta \text{treerec}\{\rho\}(x.e';e) : \rho}
\]
**Hint:** Your solution should involve recursive unfolding, a self-referential function, and generic programming.

**Solution:** Write \( \text{tree} = \text{rec}(t.\text{tree}) \). Then \( \text{treerec}\{\rho\}(x.e';e) \) is

\[
\text{fun } r(x:\text{tree}) : \rho \text{ is } (\lambda (x : 1 + (\rho \times \rho)) e')(\text{map}\{t.1 + (t \times t)\}(y.r)(\text{unfold}(x)))(e),
\]

where recursive functions are defined as above.

(c) [10 points] In FPC-by-name specify and define the generator for the infinite tree type \( \text{itree} \) represented as above.

\[
\frac{\Gamma \vdash e : \sigma \quad \Gamma, x : \sigma \vdash e' : 1 + (\sigma \times \sigma)}{\Gamma \vdash \text{itreegen}\{\sigma\}(x.e';e) : \text{itree}}
\]

**Hint:** Your solution should involve a recursive folding, a recursive definition (fixed point), and generic programming.

**Solution:** Write \( \text{itree} = \text{rec}(t.\text{itree}) \). Then \( \text{itreegen}\{\sigma\}(x.e';e) \) is

\[
\text{fold}((\text{fix } g : \sigma \rightarrow \text{itree is } \lambda (x : \sigma) \text{map}\{t.1 + (t \times t)\}(y.g(y))(e'))(e)).
\]
Appendix

<table>
<thead>
<tr>
<th>INIT</th>
<th>FINAL</th>
<th>RET</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \epsilon )</td>
<td>( \epsilon )</td>
<td>( \epsilon \rightarrow e' )</td>
</tr>
<tr>
<td>( \Sigma_m )</td>
<td>( \Sigma_{\text{ret}(e)} )</td>
<td>( \Sigma_{\text{ret}(e)} \rightarrow k )</td>
</tr>
</tbody>
</table>

**BIND**

| \( k \) | \( \Sigma \) | \( \Sigma_{\text{bnd}(e_1; x.m_2)} \rightarrow k \) |

**BIND-CMD**

| \( k \) | \( \Sigma \) | \( \Sigma \) |

**BIND-RET**

| \( e_1 \) | \( \Sigma \) | \( \Sigma_{\text{bnd}(\text{cmd}(m_1); x.m_2)} \rightarrow k \) |

**DCL**

| \( e_1 \) | \( \Sigma \) | \( \Sigma_{\text{dcl}(\tau_1)(e_1; a.m_2)} \rightarrow k \) |

**DCL-BODY**

| \( e_1 \) | \( \Sigma \) | \( \Sigma_{\text{dcl}(\tau_1)(e_1; a.m_2)} \rightarrow k \) |

**DCL-RET**

| \( e_2 \) | \( \Sigma_{a \sim \tau_1} \) | \( \Sigma_{a \sim \tau_1} \) |

**GET**

| \( k \) | \( \Sigma \) | \( \Sigma_{\text{get}[a]} \rightarrow k \) |

**SET-ARG**

| \( e \) | \( \Sigma \) | \( \Sigma_{\text{set}(a)(e)} \rightarrow k \) |

**SET**

| \( e \) | \( \Sigma \) | \( \Sigma_{\text{set}(a)(e)} \rightarrow k' \) |

Figure 1: Stack Machine for MA
datatype un = Z | S of un

type t = un

val zero = Z

val succ = S

fun plus (Z, u) = u | plus (u, Z) = u | plus (S(u), S(v)) = S(S(plus (u, v)))

fun isz (Z) = true | isz (S _) = false

Figure 2: Unary Implementation of Numbers

datatype bin = Z0 | D0 of bin | D1 of bin

type t = bin

val zero = Z0

fun succ (Z0) = D1(Z0) | succ (D0(b)) = D1(b) | succ (D1(b)) = D0(succ (b))

fun plus (Z0, b) = b | plus (b, Z0) = b | plus (D0(b'), D0(b'')) = D0(plus (b, b'')) | plus (D0(b'), D1(c'')) = D1(plus (b', c'')) | plus (D1(b'), D0(c'')) = D1(plus (b', c'')) | plus (D1(b'), D1(c'')) = D0(succ (plus (b', c'')))

fun isz (Z0) = true | isz (D0(b)) = isz (b) | isz (D1(b)) = false

Figure 3: Binary Implementation of Numbers