There are 10 pages in this examination, comprising 3 questions worth a total of 100 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 80 minutes to complete this examination.

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

There are scratch sheets at the end for your personal use. *Do not expect scratch sheet material to be graded.*
Question 1 [40]: Short Answers

(a) Labelled binary trees (with labels at their nodes) may be defined in the untyped \( \lambda \)-calculus using booleans and pairs as follows:

- \( \text{emp} \triangleq \lambda x. x \).
- \( \text{node} \triangleq \lambda l. \lambda x. \lambda r. \langle F, \langle l, (x, r) \rangle \rangle \).

Recall from HW0 that the first component of a pair is obtained by applying it to \( T \) and the second is obtained by applying it to \( F \). Define the following operations on labelled binary trees:

i. [3 points] Return \( T \) or \( F \) according to whether a tree is \( \text{emp} \):

\[
\text{isemp} \triangleq \lambda t. t \ T
\]

ii. [3 points] Left child of a node:

\[
\text{lft} \triangleq \lambda t. t \ F \ T
\]

iii. [3 points] Label of a node:

\[
\text{lbl} \triangleq \lambda t. t \ F \ F \ T
\]

iv. [3 points] Right child of a node:

\[
\text{rht} \triangleq \lambda t. t \ F \ F \ F
\]
(b) Choose the correct answer in each case.

i. [3 points] In T the value of
\[
\lambda(x:\text{nat}) \text{rec } x \{ z \mapsto \text{\textbf{1}} \mid s(y) \text{ with } z \mapsto x \times z \}
\]
is \textbf{\checkmark Itself.}  \quad \bigcirc \text{ Diverges.}  \quad \bigcirc \text{ The factorial of } x.  \quad \bigcirc \text{ None of the above.}

ii. [3 points] In PCF \textit{by-name}, the value of
\[
\text{fix } f : \text{nat} \rightarrow \text{nat} = \lambda(x:\text{nat}) \text{ifz } x \{ z \mapsto \text{\textbf{1}} \mid s(x') \mapsto x \times f(x') \}
\]
is \bigcirc \text{ Itself.}  \quad \bigcirc \text{ Diverges.}  \quad \bigcirc \text{ The underlined } \lambda\text{-abstraction.}  \quad \textbf{\checkmark None of the above.}

iii. [3 points] In PCF \textit{by-value}, the value of
\[
\text{fun } f(x:\text{nat}):\text{nat} \text{ is } \text{ifz } x \{ z \mapsto \text{\textbf{1}} \mid s(x') \mapsto x \times f(x') \}
\]
is \textbf{\checkmark Itself.}  \quad \bigcirc \text{ Diverges.}  \quad \bigcirc \text{ The factorial of } x.  \quad \bigcirc \text{ None of the above.}

iv. [3 points] In FPC \textit{by-value}, the value of
\[
\text{self } \{ \text{nat} \rightarrow \text{nat} \}(f.\lambda(x:\text{nat}) \lambda(x:\text{nat}) \text{ifz } x \{ z \mapsto \text{\textbf{1}} \mid s(x') \mapsto x \times f(x') \})
\]
is \bigcirc \text{ Diverges.}  \quad \textbf{\checkmark Itself.}  \quad \bigcirc \text{ The inner } \lambda\text{-abstraction.}  \quad \bigcirc \text{ Undefined.}
(c) Using self-reference types \texttt{self}(\tau) as defined in \textbf{FPC}-by-value, define two mutually recursive functions, \textit{even}, and \textit{odd}, of type \texttt{nat} \to \texttt{bool} according to the following informal specification:

\[
\begin{align*}
\text{even}(0) &= \text{true} \\
\text{even}(n + 1) &= \text{odd}(n) \\
\text{odd}(0) &= \text{false} \\
\text{odd}(n + 1) &= \text{even}(n)
\end{align*}
\]

Let the type \tau stand for 

\[(\texttt{nat} \to \texttt{bool}) \times (\texttt{nat} \to \texttt{bool}).\]

Define \textit{evenodd} to be the self-referential value

\[
\text{self}\{\tau\}(\texttt{eo}.\langle \texttt{ev}, \texttt{od} \rangle)
\]

of type \texttt{self}(\tau), wherein you are to provide the components \texttt{ev} and \texttt{od}, and using which you are to define \textit{even} and \textit{odd}.

i. [4 points] Define \texttt{ev} in the definition of \textit{evenodd} in terms of the variable \texttt{eo}:

\[
\lambda (x: \texttt{nat}) \text{ifz}(x; \texttt{true}; x'.\texttt{unroll}(\texttt{eo}) \cdot \texttt{r}(x'))
\]

ii. [4 points] Define \texttt{od} in the definition of \textit{evenodd} in terms of the variable \texttt{eo}:

\[
\lambda (x: \texttt{nat}) \text{ifz}(x; \texttt{false}; x'.\texttt{unroll}(\texttt{eo}) \cdot \texttt{l}(x'))
\]

iii. [4 points] Define \textit{even} in terms of \textit{evenodd}:

\[
\text{unroll}(\textit{evenodd}) \cdot \texttt{l}
\]

iv. [4 points] Define \textit{odd} in terms of \textit{evenodd}:

\[
\text{unroll}(\textit{evenodd}) \cdot \texttt{r}
\]
Question 2 [35]: Type Safety for Recursive Functions

The statics of $\text{PCF}$-by-value distinguishes two forms of typing judgment:

1. $\Gamma \vdash e : \tau$, stating that $e$ is a value of type $\tau$ in context $\Gamma$.
2. $\Gamma \vdash e \bowtie \tau$, stating that $e$ is a computation of type $\tau$ in context $\Gamma$.

Variables are values, $\Gamma, x : \tau \vdash x : \tau$, and values are computations, $\Gamma \vdash e : \tau$.

Recursive functions are values of partial function type whose bodies are computations:

\[
\Gamma, f : \tau_1 \rightarrow \tau_2, x : \tau_1 \vdash e \bowtie \tau_2 \\
\Gamma \vdash \text{fun}\{\tau_1; \tau_2\}(f.x.e) : \tau_1 \rightarrow \tau_2
\]

Applications are computations:

\[
\Gamma \vdash e_1 \bowtie \tau_2 \rightarrow \tau \quad \Gamma \vdash e_2 \bowtie \tau_2 \\
\Gamma \vdash e_1(e_2) \bowtie \tau
\]

The corresponding dynamics is given by the following rules:

\[
\text{fun}\{\tau_1; \tau_2\}(f.x.e) \text{ val} \quad e_1 \text{ val} \\
\text{ap}(\text{fun}\{\tau_1; \tau_2\}(f.x.e); e_1) \mapsto [\text{fun}\{\tau_1; \tau_2\}(f.x.e), e_1/f, x/e]
\]

Complete the following parts of the proof of type safety for $\text{PCF}$-by-value:

(a) [2 points] Canonical Form Lemmas: If $e : \tau_1 \rightarrow \tau_2$, then $e \text{ val}$ and $e = \text{fun}\{\tau_1; \tau_2\}(f.x.e)$

(b) Inversion Lemma:
   i. [2 points] If $e_1(e_2) \bowtie \tau$, then for some type $\tau_2$, $e_1 \bowtie \tau_2$ and $e_2 \bowtie \tau_2$.

   ii. [2 points] If $\text{fun}\{\tau_1; \tau_2\}(f.x.e) : \tau_1 \rightarrow \tau_2$, then $f : \tau_1 \rightarrow \tau_2, x : \tau_1 \vdash e \bowtie \tau_2$

(c) Substitution Lemma:
   i. [2 points] If $\Gamma, x : \tau \vdash e' : \tau'$, and $\Gamma \vdash e : \tau$, then $[e/x]e' : \tau$.

   ii. [2 points] If $\Gamma, x : \tau \vdash e' \bowtie \tau'$, and $\Gamma \vdash e : \tau$, then $[e/x]e' \bowtie \tau$.
(d) Preservation Theorem, case for the rule for application of a recursive function given above,

i. [2 points] Assume that \( \text{ap}(\text{fun}\{\tau_1; \tau_2\}(f.x.e); e_1) : \tau \)

ii. [2 points] We want to show that \( \text{fun}\{\tau_1; \tau_2\}(f.x.e), e_1/f, x) e : \tau \)

iii. [4 points] By the Inversion Lemma the expression \( e_1 \) has type \( \tau_1 \), and the expression \( \text{fun}\{\tau_1; \tau_2\}(f.x.e) \) has type \( \tau_1 \to \tau_2 \).

iv. [4 points] By the Inversion Lemma the expression \( e \) has type \( \tau_2 \) in the typing context \( f : \tau_1 \to \tau_2, x : \tau_1 \).

v. [3 points] Therefore by the Substitution Lemma, \( \text{fun}\{\tau_1; \tau_2\}(f.x.e), e_1/f, x) e : \tau \).

(e) Progress Theorem, case for application typing where \( e = \text{ap}(e_1; e_2) \). Consider only the case where \( e_1 \) val and \( e_2 \) val.

i. [2 points] Assume \( \Gamma \vdash \text{ap}(e_1; e_2) : \tau \)

ii. [4 points] By assumption we have \( e_1 \vdash \tau_2 \to \tau \) and \( e_2 \vdash \tau_2 \) for some type \( \tau_2 \).

iii. [3 points] By the Canonical Forms Lemma, \( e_1 \) is \( \text{fun}\{\tau_2; \tau\}(f.x.e) \).

iv. [1 point] By the dynamics of application \( \text{ap}(e_1; e_2) \mapsto \text{fun}\{\tau_2; \tau\}(f.x.e), e_2/f, x)e \).
Question 3 [25]: Polymorphic Encodings

(a) The definition of the natural numbers in $\mathbf{F}$, which makes use of sums and products that are defined in PFPL.

$$
nat \triangleq \forall (r. ((1 + r) \to r) \to r)$$

$$
natrec{\rho}(x.e')(e) \triangleq e[\rho](\lambda (x. 1 + \rho) e')$$

$$
\begin{align*}
\zeta & \triangleq \Lambda(r. \lambda (bs. (1 + r) \to r) bs(1 \cdot \langle \rangle)) \\
\sigma(e) & \triangleq \Lambda(r. \lambda (bs. (1 + r) \to r) bs(r \cdot natrec{\rho}(x.bs(x))(e)))
\end{align*}
$$

[Recall that the generic programming map construct for a type family $t.\tau$ has typing rule

$$
\Gamma, y : \rho \vdash e' : \rho' \quad \Gamma \vdash e : [\rho/t]\tau \quad
\frac{}{map\{t.\tau\}(y.e')(e) : [\rho'/t]\tau}
$$

with the dynamics defined to apply the function $\lambda (y : \rho) e'$ to the components of $e$ corresponding to occurrences of $t$ in $\tau$.]

i. [3 points] Rewrite the definition of $\sigma(e)$ given above using map for the type family $t.1 + t$:

$$
\begin{align*}
\Lambda(r. \lambda (bs. (1 + r) \to r) bs(map\{t.1 + t\}(y.natrec{\rho}(x.bs(x))(y)))(r \cdot e)
\end{align*}
$$

[Recall that if $\Gamma, x : 1 + \rho \vdash e' : \rho$, then $\Gamma, y : nat \vdash natrec{\rho}(x.e')(y) : \rho$.]

ii. [3 points] Define the consolidated “zero and successor” operation $\zeta\sigma(e) : nat$ for $\Gamma \vdash e : 1 + nat$, again using map for the family $t.1 + t$:

$$
\begin{align*}
\Lambda(r. \lambda (bs. (1 + r) \to r) bs(map\{t.1 + t\}(y.natrec{\rho}(x.bs(x))(y)))(e))
\end{align*}
$$
(b) The type \( \text{dfa}_\Sigma \) from assignment 3 can be defined in \( \text{F} \) using existential types by packaging the current state along with the state table.

\[
\text{dfa}_\Sigma \triangleq \exists (t.t \times (t \rightarrow 2 \times (\Sigma \rightarrow t)))
\]

The existential type hides the choice of state type from the outside.

i. [4 points] Define function \( \text{new}_\Sigma : \forall (Q.Q \rightarrow \text{spec}_\Sigma \rightarrow \text{dfa}_\Sigma) \), where \( \text{spec}_\Sigma \triangleq Q \rightarrow (2 \times (\Sigma \rightarrow Q)) \). \( \text{new}_\Sigma \) forms a \( \text{dfa}_\Sigma \) for any choice of \( Q \) and corresponding state table \( \text{spec}_\Sigma \). Note that \( \text{new}_\Sigma \) is now polymorphic in terms of the choice of state type \( Q \).

\[
\text{new}_\Sigma \triangleq \ldots
\]

\( \Lambda Q. \lambda (q_0 : Q) \lambda (\delta : \text{spec}_\Sigma) \text{pack } Q \text{ with } \langle q_0, \delta \rangle \text{ as } \text{dfa}_\Sigma \)

ii. [3 points] Define the function \( \text{accept}_\Sigma : \text{dfa}_\Sigma \rightarrow 2 \) that returns whether the machine is in an accepting state:

\[
\text{accept}_\Sigma \triangleq \lambda (m : \text{dfa}_\Sigma) \ldots
\]

\( \text{open } m \text{ as } s \text{ with } \langle q, \delta \rangle \text{ in } \delta(q) : 1 \)

iii. [5 points] Define the function \( \text{step}_\Sigma : \text{dfa}_\Sigma \rightarrow (\Sigma \rightarrow \text{dfa}_\Sigma) \) that steps a machine according to a character. (Hint: \( \text{map} \) will significantly simplify your solution if used properly.)

\[
\text{open } m \text{ as } s \text{ with } \langle q, \delta \rangle \text{ in } \text{map}(t.\Sigma \rightarrow t)(\text{new}_\Sigma[s](q)(\delta))(\delta(q) \cdot r)
\]

(c) [7 points] A product machine \( m_1 \times m_2 \) is formed by combining two \( \text{dfa}_\Sigma \)s, namely \( m_1 \) and \( m_2 \). \( m_1 \times m_2 \) accepts a string if and only if both machines accept the string. A function \( \text{all} : 2 \rightarrow 2 \rightarrow 2 \) that returns \( \text{accept} \) iff both arguments are \( \text{accept} \) is already made available to you. Define \( m_1 \times m_2 \) in terms of \( m_1 \) and \( m_2 \).

**Solution:**

\[
\text{open } m_1 \text{ as } s_1 \text{ with } \langle q_1, \delta_1 \rangle \text{ in open } m_2 \text{ as } s_2 \text{ with } \langle q_2, \delta_2 \rangle \text{ in } \text{new}_\Sigma[s_1 \times s_2](\langle q_1, q_2 \rangle)(\delta_{12})
\]

where \( \langle a_1, t_1 \rangle = \delta_1(q_1), \langle a_2, t_2 \rangle = \delta_2(q_2) \),

\[
\delta_{12} \triangleq \lambda (\langle q_1, q_2 \rangle : s_1 \times s_2) (\text{all}(a_1)(a_2), \lambda (\sigma : \Sigma) (t_1(\sigma), t_2(\sigma)))
\]