1 Exceptions [60 pts]

Consider the abstract machine \( C \) for the simply typed \( \lambda \)-calculus:

\[
\begin{align*}
\text{type} & : A ::= P \mid A \to A \\
\text{base type} & : P \\
\text{expression} & : e ::= x \mid \lambda x : A. e \mid e \ e \\
\text{value} & : v ::= \lambda x : A. e \\
\text{frame} & : \phi ::= \square \ e \ \mid \ v \\
\text{stack} & : \sigma ::= \square \mid \sigma ; \phi \\
\text{state} & : s ::= \sigma \ \triangleright \ e \ \mid \sigma \ \triangleleft \ v
\end{align*}
\]

The goal of this problem is to extend the abstract machine \( C \) with exceptions.

To begin with, we introduce two new forms of expressions \( \text{try} \ e \) with \( e' \) and \( \text{exn} \):

\[
\begin{align*}
\text{expression} & : e ::= \cdots \mid \text{try} \ e \ \text{with} \ e' \mid \text{exn}
\end{align*}
\]

Informally \( \text{try} \ e \ \text{with} \ e' \) tries to evaluate \( e \). If \( e \) successfully evaluates to \( v \), then \( \text{try} \ e \ \text{with} \ e' \) also evaluates to the same value \( v \). In this case, \( e' \) is never visited and is thus ignored. If the evaluation of \( e \) raises an exception by attempting to reduce \( \text{exn} \), then the evaluation of \( e \) is canceled and \( e' \) is evaluated instead. In this way, \( \text{try} \ e \ \text{with} \ e' \) handles every exception raised within \( e \). Note that \( e' \) itself may also raise an exception, in which case the exception propagates to the next \( \text{try} \ e'_{\text{next}} \) with \( e'_{\text{next}} \) such that \( e'_{\text{next}} \) encloses \( \text{try} \ e \ \text{with} \ e' \).

Formally we extend the operational semantics with the following reduction rules.

\[
\begin{align*}
\text{exn} \ e \ \mapsto & \text{exn} \\
\text{exn} \ e \ e & \mapsto \text{exn}' \\
\text{try} \ e_1 \ \text{with} \ e_2 \ & \mapsto \text{try} \ e'_1 \ \text{with} \ e_2 \\
\text{try} \ v \ e_1 \ & \mapsto v \\
\text{try} \ e_1 \ \text{with} \ e_2 \ & \mapsto \text{try} \ e_1 \ \text{with} \ e_2
\end{align*}
\]

Give new rules for the reduction judgment \( s \ \mapsto_C \ s' \) corresponding to the above five reduction rules. You have to use an additional state \( \sigma \ \ll\ll \ \text{exn} \):

\[
\begin{align*}
\text{state} & : s ::= \cdots \mid \sigma \ \ll\ll \ \text{exn}
\end{align*}
\]

- A state \( \sigma \ \ll\ll \ \text{exn} \) means that the machine is currently propagating an exception \( \text{exn} \).

Instruction:

1. You will need exactly five rules.

2. Write only those rules related with \( \text{try} \ e \ \text{with} \ e' \) and \( \text{exn} \). For example, do not copy the rules from the course notes.

\[
\begin{align*}
\sigma \ \triangleright \ \text{try} \ e_1 \ \text{with} \ e_2 \ \mapsto_C \ \sigma ; \text{try} \ \square \ \text{with} \ e_2 \ \triangleright \ e_1 \\
\sigma ; \text{try} \ \square \ \text{with} \ e_2 \ \ll \ v \ \mapsto_C \ \sigma \ \ll \ v \\
\sigma \ \triangleright \ \text{exn} \ \mapsto_C \ \sigma \ \ll\ll \ \text{exn}
\end{align*}
\]
Consider the following simply typed λ-calculus extended with recursive types.

\[
\text{type } A ::= \text{unit} | A \to A | \alpha | \mu \alpha. A
\]

\[
\text{expression } e ::= x | \lambda x : A. e | e \ e | () | \text{fold}_C e | \text{unfold}_C e
\]

**Question 1.** [20 points] Define the call-by-value operational semantics of those constructs for recursive types. You have to write only those rules related with \(\text{fold}_C e\) and \(\text{unfold}_C e\). You will need exactly three rules.

\[
\begin{align*}
\text{Fold} & \quad e \mapsto e' \\
\text{fold}_C e & \mapsto \text{fold}_C e'
\end{align*}
\]

\[
\begin{align*}
\text{Unfold} & \quad e \mapsto e' \\
\text{unfold}_C e & \mapsto \text{unfold}_C e'
\end{align*}
\]

\[
\begin{align*}
\text{Unfold}^2 & \quad \text{unfold}_C \text{fold}_C v \mapsto v
\end{align*}
\]

**Question 2.** [20 points] State the reason why we do not need a reduction rule for \(\text{fold}_C \text{unfold}_C v\) (where \(v\) is a value):

If \(v = \text{fold}_C v'\) for a value \(v'\), then

\[\text{fold}_C \text{unfold}_C v = \text{fold}_C \text{unfold}_C \text{fold}_C v' \mapsto \text{fold}_C v'\]

Otherwise, \(\text{fold}_C \text{unfold}_C v\) get stuck. Therefore, we do not need any rule for \(\text{fold}_C \text{unfold}_C v\).