1 Checked Error Handling

In this exercise we use the Simply-Typed Lambda Calculus (STLC) extended with rules for error handling. In this language, terms may reduce to a normal form `error`, which is *not* a value. In addition, we add the new term form `try` $\mathtt{t}_1$ `with` $\mathtt{t}_2$, which allows handling errors that occur while evaluating $\mathtt{t}_1$.

Here is a summary of the extensions to syntax and evaluation:

$$\begin{array}{rcll} t & ::= & & \textbf{terms}: \\ & | & \ldots & \\ & | & \texttt{error} & \text{run-time error} \\ & | & \texttt{try t with t} & \text{trap errors} \end{array}$$

New evaluation rules:

$$\texttt{error t}_2 \longrightarrow \texttt{error}\ (\text{E-AppErr1}) \qquad \texttt{try v}_1\ \texttt{with t}_2 \longrightarrow \texttt{v}_1\ (\text{E-TryValue})$$

$$\texttt{v}_1\ \texttt{error} \longrightarrow \texttt{error}\ (\text{E-AppErr2}) \qquad \texttt{try error with t}_2 \longrightarrow \texttt{t}_2\ (\text{E-TryError})$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{try t}_1\ \texttt{with t}_2 \longrightarrow \texttt{try t}_1'\ \texttt{with t}_2}\ (\text{E-Try})$$

These extensions are exactly those summarized in Figures 14-1 and 14-2 on pages 172 and 174 of the TAPL book. However, we will use *different* typing rules, because we want a stronger progress result.

Unlike the relaxed progress statement used by TAPL, we want to ensure that all errors are eventually caught, i.e., that no error reaches the top-level. For example, we want to accept `try error with true` but we want to reject $(\lambda\texttt{x:Bool.x})$ `error`.

The goal of this exercise is to prove that the following progress theorem holds:

Theorem 1 (Progress)**.** *If* $\emptyset$; $\texttt{false} \vdash \texttt{t:T}$*, then either* $\texttt{t}$ *is a value or else* $\texttt{t} \longrightarrow \texttt{t}'$*.*

The above theorem uses a typing judgment extended with a Boolean value E, written

$$\Gamma\ ;\ E \vdash \texttt{t:T}$$

where $E \in \{\texttt{true}, \texttt{false}\}$.

The theorem says that a well-typed term that is closed (that is, it does not have free variables, which is expressed using $\Gamma = \emptyset$) is either a value, or else it can be reduced *as long as* $E = \texttt{false}$.

Intuitively, the predicate E in the typing judgment $\Gamma\ ;\ E \vdash \texttt{t:T}$ determines whether the term $\texttt{t}$ is in an *impure* position, i.e. whether it is allowed to reduce to `error` or not.

The typing rules inherited from STLC "push" the impurity predicate E unchanged from the conclusion into the premises. Intuitively, if a term is allowed to produce an error, all its sub-terms are also allowed to do so.

$$\frac{\texttt{x} : \texttt{T} \in \Gamma}{\Gamma\ ;\ E \vdash \texttt{x} : \texttt{T}}\ (\text{T-Var}) \qquad \frac{\Gamma,\ \texttt{x} : \texttt{T}_1\ ;\ E \vdash \texttt{t}_2 : \texttt{T}_2}{\Gamma\ ;\ E \vdash \lambda\texttt{x} : \texttt{T}_1.\texttt{t}_2 : \texttt{T}_1 \to \texttt{T}_2}\ (\text{T-Abs})$$

$$\frac{\Gamma\ ;\ E \vdash \texttt{t}_1 : \texttt{T}_1 \to \texttt{T}_2 \quad \Gamma\ ;\ E \vdash \texttt{t}_2 : \texttt{T}_1}{\Gamma\ ;\ E \vdash \texttt{t}_1\ \texttt{t}_2 : \texttt{T}_2}\ (\text{T-App})$$

Your task is as follows:

1. Design typing rules for `error` (T-Error) and `try t with t` (T-Try) terms, so that you are able to prove the progress theorem (and so that it is possible to write reasonable programs).

2. Prove the above progress theorem using structural induction. (You can use the canonical forms lemma for STLC as seen in the lecture without proof.)

3. How does E contribute to your proof? Where would your proof "go wrong" if we removed it from the typing rules?

4. *(Extra effort)* Try to prove the preservation theorem as well. Is it possible with the typing rules you designed? If not, is there a way to alter the typing rules to make the theorem hold? In either case, it may be useful to slightly change the form of the function type.

2 The call-by-value simply typed lambda calculus with returns

Consider a variant of the call-by-value simply typed lambda calculus extended to support a new language construct: `return t`, which immediately returns a given term `t` from an **enclosing** lambda, disregarding any potential further computation typically needed for call-by-value evaluation rules.

This system extends STLC with booleans with two new forms of terms:

$$\begin{array}{rcll} \mathtt{t} & ::= & \cdots & \textbf{terms}: \\ & | & \mathtt{return\ t} & \text{return} \\ & | & \mathtt{frame\ t} & \text{frame} \end{array}$$

`frame` is not meant to be used in terms representing source programs. Rather, it is introduced with the following reduction rule:

$$(\lambda \mathtt{x{:}T_1}.\mathtt{t_{12}})\ \mathtt{v} \longrightarrow \mathtt{frame}\ ([\mathtt{x} \mapsto \mathtt{v}]\mathtt{t_{12}}) \qquad \text{(E-App)}$$

The frame form marks the scope from which local returns should return. If the body of a frame reduces to a value, the frame should be removed. Otherwise, a return form should stop propagating at a frame in much the same way that an error stops at a `try/with`.

In the previous exercise, we extended the typing rules to track impurity. We do something similar here, but to track the types of values that can be *returned* from a term. That is in addition to the normal type of the term, which is the one we get if it finishes evaluating without encountering a **return**. We use `R` to denote a *set* of types, i.e., $\{\mathtt{T}_1, \ldots, \mathtt{T}_n\}$.

$$\Gamma \vdash \mathtt{true} : \mathtt{Bool} \mid \emptyset \qquad \text{(T-True)}$$

$$\Gamma \vdash \mathtt{false} : \mathtt{Bool} \mid \emptyset \qquad \text{(T-False)}$$

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{T_1} \to \mathtt{T_2} \mid \mathtt{R_1} \quad \Gamma \vdash \mathtt{t_2} : \mathtt{T_1} \mid \mathtt{R_2}}{\Gamma \vdash \mathtt{t_1\ t_2} : \mathtt{T_2} \mid (\mathtt{R_1} \cup \mathtt{R_2})} \qquad \text{(T-App)}$$

$$\frac{\mathtt{x} : \mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T} \mid \emptyset} \qquad \text{(T-Var)}$$

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{T_1} \mid \mathtt{R_1}}{\Gamma \vdash \mathtt{return\ t_1} : \mathtt{T} \mid (\mathtt{R_1} \cup \{\mathtt{T_1}\})} \qquad \text{(T-Return)}$$

Note that the conclusion of T-Return mentions a `T` that does not appear anywhere in the premises. What does that mean?

In this exercise, you are to given typing rules for abstractions and frames, and adjust the existing evaluation rules such that they correctly and comprehensively describe the semantics of the extension. In particular, **return** terms must short-circuit most of the terms, much like **error** terms did in the previous excercise.

You must then prove the preservation theorem for the system.

More precisely, you have the following tasks:

1. Extend the evaluation rules (by adding new rules and/or changing existing ones) to express the early return semantics provided by return. Identify the evaluation strategy used by the specification and make sure that your extension adheres to it. Make sure that `return` terms propagate like `error` terms did in the previous exercise, so that progress would hold.

2. Design typing rules for abstractions (T-Abs) and frames (T-Frame) so that you are able to prove the following lemmas and preservation theorem (and so that it is possible to write reasonable programs).

3. Prove the following two lemmas:

   Values are pure: If $\Gamma \vdash \texttt{v} : \texttt{T} \mid \texttt{R}$, then $\texttt{R} = \emptyset$.

   return is polymorphic: If $\Gamma \vdash \texttt{return t}_1 : \texttt{T} \mid \texttt{R}$, then $\Gamma \vdash \texttt{return t}_1 : \texttt{U} \mid \texttt{R}$.

4. State, but do not prove, the weakening lemma, the permutation lemma, and the substitution lemma for this system.

5. State and prove the preservation theorem. You may rely on the above lemmas as well as the usual inversion lemma and canonical forms lemma. Think about how `R` evolves when we take one step of evaluation: does it always remain the same?

6. *(Extra effort)* State and prove the progress theorem as well. The statement should exclude top-level returns – intuitively, a closed term with a "naked" return doesn't make sense.

Before you begin, think carefully about the following simple term: λ`x:Bool.(return true) x`. Intuitively, it makes sense. Once this lambda is applied, it is going to evaluate to `true`, regardless of the input. What type or types will be assigned to `return true`?

3 Appendix

3.1 The call-by-value simply typed lambda calculus with booleans

The complete reference of the variant of simply typed lambda calculus (with *Bool* ground type representing the type of values *true* and $false$) used in "The call-by-value simply typed lambda calculus with returns" is as follows:

$\mathtt{t}$	$::=$		**terms** :
	$\mid$	$\mathtt{x}$	variable
	$\mid$	$\lambda \mathtt{x}{:}\mathtt{T}.\mathtt{t}$	abstraction
	$\mid$	$\mathtt{t}\ \mathtt{t}$	application
	$\mid$	$\mathtt{true} \mid \mathtt{false}$	boolen values
	$\mid$	$\mathtt{if\ t\ then\ t\ else\ t}$	conditional
$\mathtt{v}$	$::=$		**values** :
	$\mid$	$\lambda \mathtt{x}{:}\mathtt{T}.\mathtt{t}$	abstraction
	$\mid$	$\mathtt{true} \mid \mathtt{false}$	boolen values
$\mathtt{T}$	$::=$		**types** :
	$\mid$	$\mathtt{T} \to \mathtt{T}$	function type
	$\mid$	$\mathtt{Bool}$	boolean type

Evaluation rules:

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2}\ \text{(E-App1)} \qquad \mathtt{if\ true\ then\ t_2\ else\ t_3} \longrightarrow \mathtt{t_2}\ \text{(E-IfTrue)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1\ \mathtt{t}_2'}\ \text{(E-App2)} \qquad \mathtt{if\ true\ then\ t_2\ else\ t_3} \longrightarrow \mathtt{t_3}\ \text{(E-IfFalse)}$$

$$(\lambda \mathtt{x}{:}\mathtt{T}.\mathtt{t}_{12})\ \mathtt{v}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12}\ \text{(E-AppAbs)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{if\ t_1\ then\ t_2\ else\ t_3} \longrightarrow \mathtt{if\ t_1'\ then\ t_2\ else\ t_3}}\ \text{(E-If)}$$

Typing rules:

$$\frac{\mathtt{x}{:}\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x}{:}\mathtt{T}}\ \text{(T-Var)} \qquad \Gamma \vdash \mathtt{true{:}Bool}\ \text{(T-True)}$$

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2{:}\mathtt{T}_2}{\Gamma \vdash (\lambda \mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_2){:}\mathtt{T}_1 \to \mathtt{T}_2}\ \text{(T-Abs)} \qquad \Gamma \vdash \mathtt{false{:}Bool}\ \text{(T-False)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1{:}\mathtt{T}_1 \to \mathtt{T}_2 \qquad \Gamma \vdash \mathtt{t}_2{:}\mathtt{T}_1}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2{:}\mathtt{T}_2}\ \text{(T-App)} \qquad \frac{\Gamma \vdash \mathtt{t}_1{:}\mathtt{Bool} \qquad \Gamma \vdash \mathtt{t}_2{:}\mathtt{T} \qquad \Gamma \vdash \mathtt{t}_3{:}\mathtt{T}}{\Gamma \vdash \mathtt{if\ t_1\ then\ t_2\ else\ t_3{:}T}}\ \text{(T-If)}$$