MPRI course 2-4
"Functional programming languages"
Answers to the exercises

Xavier Leroy

Part I: Operational semantics

Exercise I.1 Note that terms that can reduce are necessarily applications $a = a_1\ a_2$. This is true for head reductions (the β_v rule) and extends to reductions under contexts because non-trivial contexts are also applications. Since values are not applications, it follows that values do not reduce.

Now, assume $a = E_1[a_1] = E_2[a_2]$ where a_1 and a_2 reduce by head reduction and E_1, E_2 are evaluation contexts. We show $E_1 = E_2$ and $a_1 = a_2$ by induction over the structure of a. By the previous remark, a must be an application $b\ c$. We argue by case on whether b or c are applications.

- Case 1: b is an application. b is not a λ-abstraction, so a cannot head-reduce by β_v, and therefore we cannot have $E_i = [\,]$ for $i = 1, 2$. Similarly, b is not a value, therefore we cannot have $E_i = b\ E'_i$. The only case that remains possible is $E_i = E'_i\ c$ for $i = 1, 2$. We therefore have two decompositions $b = E'_1[a_1] = E'_2[a_2]$. Applying the induction hypothesis to b, which is a strict subterm of a, it follows that $a_1 = a_2$ and $E'_1 = E'_2$, and therefore $E_1 = E_2$ as well.
- Case 2: b is not an application but c is. b cannot reduce, so the case $E_i = E'_i\ c$ is impossible. c is not a value, so the case $E_i = [\,]$ is also impossible. The only possibility is therefore that b is a value and $E_i = b\ E'_i$. The result follows from the induction hypothesis applied to c and its two decompositions $c = E'_1[a_1] = E'_2[a_2]$.
- Case 3: neither b nor c are applications. The only possibility is $E_1 = E_2 = [\,]$ and $a_1 = a_2 = a$.

Exercise I.2 For each proposed rule $a \to b$, we expand the derived forms in a (written $\approx$ below), perform reductions with the rules for the core constructs, then reintroduce derived forms in the result when necessary. For the `let` rule, this gives:

$$(\texttt{let}\ x = v\ \texttt{in}\ a) \approx (\lambda x.a)\ v \to a[x \leftarrow v]$$

by β_v-reduction. For `if`/`then`/`else`:

$$\begin{array}{rcl}
\texttt{if true then}\ a\ \texttt{else}\ b & \approx & \texttt{match True() with True()} \to a \mid \texttt{False()} \to b \\
& \to & a \\
\texttt{if false then}\ a\ \texttt{else}\ b & \approx & \texttt{match False() with True()} \to a \mid \texttt{False()} \to b \\
& \to & \texttt{match False() with False()} \to b \\
& \to & b
\end{array}$$

by `match`-reduction. Note that the second rule actually corresponds to two reductions in the base language. Finally, for pairs and projections:

$$\begin{array}{lclcl}
\texttt{fst}(v_1, v_2) & \approx & (\texttt{match Pair}(v_1, v_2) \texttt{ with Pair}(x_1, x_2) \to x_1) & \to & x_1[x_1 \leftarrow v_1, x_2 \leftarrow v_2] = v_1 \\
\texttt{snd}(v_1, v_2) & \approx & (\texttt{match Pair}(v_1, v_2) \texttt{ with Pair}(x_1, x_2) \to x_2) & \to & x_2[x_1 \leftarrow v_1, x_2 \leftarrow v_2] = v_2
\end{array}$$

again by `match` reductions.

Exercise I.3 Assume $1\ 2 \Rightarrow v$ for some v. There is only one evaluation rule that can conclude this:

$$\frac{1 \Rightarrow \lambda x.c \qquad 2 \Rightarrow v' \qquad c[x \leftarrow v'] \Rightarrow v}{1\ 2 \Rightarrow v}$$

but of course 1 evaluates only to 1 and not to any λ-abstraction.

Now, assume that we have a derivation $a' \Rightarrow v$. By examination of the rules that can conclude this derivation, it can only be of the following form:

$$\frac{\lambda x.x \Rightarrow \lambda x.x \qquad \lambda x.x \Rightarrow \lambda x.x \qquad \dfrac{\vdots}{(x\ x)[x \leftarrow \lambda x.x] = a' \Rightarrow v}}{(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \Rightarrow v}$$

Therefore, any derivation D of $a' \Rightarrow v$ contains a sub-derivation D' of $a' \Rightarrow v$ that is strictly smaller than D. Since derivations for the $\Rightarrow$ predicate are finite, this is impossible.

The difference between these two examples is visible on their reduction sequences: a is an erroneous evaluation (a term that does not reduce but is not a value), while a' reduces infinitely. The evaluation relation does not hold in either of these two cases.

Exercise I.4 The base case for the induction is $a = (\lambda x.c)\ v' \to c[x \leftarrow v'] = b$. We can build the following derivation of $a \Rightarrow v$ from that of $b \Rightarrow v$:

$$\frac{\lambda x.c \Rightarrow \lambda x.c \qquad v' \Rightarrow v' \qquad c[x \leftarrow v'] = b \Rightarrow v}{a = (\lambda x.c)\ v' \Rightarrow v}$$

using the fact that $v' \Rightarrow v'$ for all values v' (check it by case over v').

The first inductive case is $a = a'\ c \to b'\ c = b$ where $a' \to b'$. The evaluation derivation for $b \Rightarrow v$ is of the following form:

$$\frac{b' \Rightarrow \lambda x.d \qquad c \Rightarrow v' \qquad d[x \leftarrow v'] \Rightarrow v}{b'\ c \Rightarrow v}$$

Applying the induction hypothesis to the reduction $a' \to b'$ and the evaluation $b' \Rightarrow \lambda x.d$, it follows that $a' \Rightarrow \lambda x.d$. We can therefore build the following derivation:

$$\frac{a' \Rightarrow \lambda x.d \qquad c \Rightarrow v' \qquad d[x \leftarrow v'] \Rightarrow v}{a'\ c \Rightarrow v}$$

which concludes $a \Rightarrow v$ as claimed.

The second inductive case is $a = v'\ a' \to v'\ b' = b$ where $a' \to b'$. The evaluation derivation for $b \Rightarrow v$ is of the following form:

$$\frac{v' \Rightarrow \lambda x.c \qquad b' \Rightarrow v'' \qquad c[x \leftarrow v''] \Rightarrow v}{v'\ b' \Rightarrow v}$$

Applying the induction hypothesis to the reduction $a' \to b'$ and the evaluation $b' \Rightarrow v''$, it follows that $a' \Rightarrow v''$. We can therefore build the following derivation:

$$\frac{v' \Rightarrow \lambda x.c \qquad a' \Rightarrow v'' \qquad c[x \leftarrow v''] \Rightarrow v}{v'\ a' \Rightarrow v}$$

which concludes $a \Rightarrow v$ as claimed.

Exercise I.5 A convenient representation for contexts E is as Caml functions taking a term a and returning the term $E[a]$.

```
type context = term -> term

let top : context = fun x -> x

let appleft (c: context) (b: term) : context = fun x -> App(c x, b)

let appright (a: term) (c: context) : context = fun x -> App(a, c x)
```

The decomposition of a term a into a context and a subterm that potentially reduces follows the same reasoning as in exercise I.1. The base cases are 1- a is not an application, and 2- a is an application of a value to a value. In these cases, the context must be the "top" context. Otherwise, we have a application $a = a_1\ a_2$ and we hunt for a potential redex in a_1, unless a_1 is already a value in which case we should look into a_2.

```
let rec decomp = function
  | App(a, b) ->
      if isvalue a then
        if isvalue b then
          (top, App(a, b))
        else
          let (c, b') = decomp b in (appright a c, b')
      else
        let (c, a') = decomp a in (appleft c b, a')
  | a ->
      (top, a)
```

Reductions at head and under contexts:

```
let head_reduce = function
  | App(Lam(x, a), v) when isvalue v -> Some(subst x v a)
  | _ -> None

let reduce a =
  let (c, a') = decomp a in
  match head_reduce a' with
  | Some a'' -> Some (c a'')
  | None -> None
```

Iterated reductions:

```
let rec evaluate a =
  match reduce a with None -> a | Some a' -> evaluate a'
```

Concerning efficiency, this interpreter has the same (bad) complexity as the SOS-based interpreter from the lecture. It is slightly less efficient in practice because the context must be explicitly constructed by `decomp`, then applied in `reduce`. Instead, the SOS-based interpreter combines the three phases (decompose, head-reduce, reconstruct by applying context) in one single traversal.

Exercise I.6 For question 1, define $I = \lambda x.x$ and take $a = (I\ I)\ (I\ I)$. We can reduce on the left of the top-level application to $a_1 = I\ (I\ I)$. But we can also reduce on the right, obtaining $a_2 = (I\ I)\ I$.

For question 2, the reduction sequences built during the proof of theorem 3 happen to use only left-to-right reductions, but remain valid with non-deterministic reductions. Concerning theorem 4, the proof of the second inductive case (see exercise I.4) never uses the hypothesis that the left part of the application is a value, therefore it remains valid if the reduction rule (app-r) is replaced by (app-r'). We therefore have the following equivalences:

$$\begin{array}{ll} & a \xrightarrow{*} v \text{ with the left-to-right evaluation strategy} \\ \text{if and only if} & a \Rightarrow v \\ \text{if and only if} & a \xrightarrow{*} v \text{ with the non-deterministic evaluation strategy.} \end{array}$$

Question 3: in light of question 2, we must look for a term that does not evaluate to a value, but instead diverges or causes an error. An example is $a = (1\ 2)\ \omega$, where ω is a term that diverges. With left-to-right reductions, a cannot reduce and is not a value, therefore its evaluation terminates immediately on an error. With non-deterministic reductions, we can choose to reduce infinitely often in ω, the argument part of the top-level evaluation, therefore observing divergence.

Part II: Abstract machines

Exercise II.1

$$\begin{array}{rcl} \mathcal{N}(\underline{n}) & = & \texttt{ACCESS}(n); \texttt{APPLY} \\ \mathcal{N}(\lambda.a) & = & \texttt{GRAB}; \mathcal{N}(a) \\ \mathcal{N}(a\ b) & = & \texttt{CLOSURE}(\mathcal{N}(b)); \mathcal{N}(a) \end{array}$$

We represent function arguments and values of variables by zero-argument closures, i.e. thunks. The `ACCESS` instruction of Krivine's machine is simulated in the ZAM by an `ACCESS` (which fetches the thunk associated with the variable) followed by an `APPLY` (which jumps to this thunk, forcing its evaluation). The `GRAB` ZAM instruction behaves like the `GRAB` of Krivine's machine if we never push a mark on the stack, which is the case in the compilation scheme above. Finally, the `PUSH` instruction of Krivine's machine and the `CLOSURE` instruction of the ZAM behave identically.

Exercise II.2 Quite simply:

$$\begin{array}{rcl} \mathcal{C}(\underline{n},\ k) & = & \texttt{ACCESS}(n); k \\ \mathcal{C}(\lambda.a,\ k) & = & \texttt{CLOSURE}(\texttt{GRAB}; \mathcal{T}(a)); k \\ \mathcal{C}(\texttt{let}\ a\ \texttt{in}\ b,\ k) & = & \mathcal{C}(a,\ \texttt{GRAB}; \mathcal{C}(b,\ \texttt{ENDLET}; k)) \\ \mathcal{C}(a\ a_1\ \dots\ a_n,\ k) & = & \texttt{PUSHRETADDR}(k); \mathcal{C}(a_n,\ \dots\ \mathcal{C}(a_1,\ \mathcal{C}(a,\ \texttt{APPLY}))) \end{array}$$

The $\mathcal{T}$ schema is adjusted accordingly:

$$\begin{array}{rcl} \mathcal{T}(\lambda.a) & = & \texttt{GRAB}; \mathcal{T}(a) \\ \mathcal{T}(\texttt{let}\ a\ \texttt{in}\ b) & = & \mathcal{C}(a,\ \texttt{GRAB}; \mathcal{T}(b)) \\ \mathcal{T}(a\ a_1\ \dots\ a_n) & = & \mathcal{C}(a_n,\ \dots\ \mathcal{C}(a_1,\ \mathcal{T}(a))) \\ \mathcal{T}(a) & = & \mathcal{C}(a,\ \texttt{RETURN}) \qquad \text{(otherwise)} \end{array}$$

Exercise II.3 At the level of the instruction set, we can add a $\texttt{COND}(c_1, c_2)$ instruction that tests the boolean value at the top of the stack and continues execution with one of two possible instruction sequences, c_1 if the boolean is `true`, c_2 otherwise. The transitions for this new instruction can be:

Machine state before			Machine state after		
Code	Env	Stack	Code	Env	Stack
$\texttt{COND}(c_1, c_2); c$	e	$\texttt{true}.s$	c_1	e	s
$\texttt{COND}(c_1, c_2); c$	e	$\texttt{false}.s$	c_2	e	s

In the compilation scheme, the translation of `if/then/else` in tail-call position is straightforward:

$$\mathcal{T}(\texttt{if}\ a\ \texttt{then}\ a_1\ \texttt{else}\ a_2) \quad = \quad \mathcal{C}(a,\ \texttt{COND}(\mathcal{T}(a_1), \mathcal{T}(a_2)))$$

An `if/then/else` in non-tail-call position is more delicate. The naive approach just duplicates the continuation code k in both arms of the conditional:

$$\mathcal{C}(\texttt{if}\ a\ \texttt{then}\ a_1\ \texttt{else}\ a_2,\ k) \quad = \quad \mathcal{C}(a,\ \texttt{COND}(\mathcal{C}(a_1, k), \mathcal{C}(a_2, k)))$$

However, this can cause code size explosion if many conditionals are nested. Another approach uses PUSHRETADDR and RETURN to share the continuation code k between both branches:

$$\mathcal{C}(\texttt{if } a \texttt{ then } a_1 \texttt{ else } a_2,\ k) \quad = \quad \texttt{PUSHRETADDR}(k); \mathcal{C}(a,\ \texttt{COND}(\mathcal{C}(a_1, \texttt{RETURN}), \mathcal{C}(a_2, \texttt{RETURN})))$$

Yet another solution modifies the dynamic semantics (the transition rule) for COND, so that the code c that follows the COND is not discarded, but magically appended to whatever arm is taken:

Machine state before			Machine state after		
Code	Env	Stack	Code	Env	Stack
$\texttt{COND}(c_1, c_2); c$	e	$\texttt{true}.s$	$c_1.c$	e	s
$\texttt{COND}(c_1, c_2); c$	e	$\texttt{false}.s$	$c_2.c$	e	s

In this case, compilation without code duplication is straightforward:

$$\mathcal{C}(\texttt{if } a \texttt{ then } a_1 \texttt{ else } a_2,\ k) \quad = \quad \mathcal{C}(a,\ \texttt{COND}(\mathcal{C}(a_1, \varepsilon), \mathcal{C}(a_2, \varepsilon)); k)$$

However, it looks like the machine is generating new code sequences on the fly during execution, which is not very realistic. To address this issue, "real" abstract machines (like Caml's or Java's) introduce conditional and unconditional branch instructions that skip over a given number of instructions.

Exercise II.4 Since the machine state decompiles to a, the machine state is of the form

$$\begin{aligned} \text{code} &= \mathcal{C}(a') \\ \text{env} &= \mathcal{C}(e') \\ \text{stack} &= \mathcal{C}(a_1[e_1] \ldots a_n[e_n]) \end{aligned}$$

and $a = a'[e']\ a_1[e_1]\ \ldots\ a_n[e_n]$.

Since the machine is stopped (cannot make a transition), we are in one of the two following cases:

1. A GRAB instruction on an empty stack, meaning that $n = 0$ and $\mathcal{C}(a') = \texttt{GRAB}; c$ for some code c. By examination of the compilation scheme, it follows that $a' = \lambda a''$. Therefore, $a = (\lambda a'')[e']$ is a value.

2. An $\texttt{ACCESS}(m)$ instruction where $\mathcal{C}(e')(m)$ is undefined. By examination of the compilation scheme, it follows that $a' = \underline{m}$. Therefore, $a = \underline{m}[e']\ \ a_1[e_1]\ \ldots\ a_n[e_n]$ is not a value and cannot reduce, since $\underline{m}[e']$ cannot reduce ($e'(m)$ is undefined).

Exercise II.5 We write $\mathcal{D}(c, S) = a$ to mean that the symbolic machine, started in code c and symbolic stack S, stops on the configuration $(\varepsilon, a.\varepsilon)$. By definition of the transitions of the symbolic machine, this partial function $\mathcal{D}$ satisfies the following equations:

$$\begin{aligned} \mathcal{D}(\varepsilon, a.\varepsilon) &= a \\ \mathcal{D}(\texttt{CONST}(N).c, S) &= \mathcal{D}(c, N.S) \\ \mathcal{D}(\texttt{ADD}.c, b.a.S) &= \mathcal{D}(c, (a+b).S) \\ \mathcal{D}(\texttt{SUB}.c, b.a.S) &= \mathcal{D}(c, (a-b).S) \end{aligned}$$

By definition of decompilation, the concrete machine state (c, s) decompiles to a iff $\mathcal{D}(c, s) = a$.

We start by the following technical lemma that shows the compatibility between symbolic execution and reduction of one expression contained in the symbolic stack.

Lemma 1 (Compatibility) *Let s be a stack of integer values, a an expression and S a stack of expressions. Assume that $\mathcal{D}(c, S.a.s) = r$ and that $a \to a'$. Then, there exists r' such that $\mathcal{D}(c, S.a'.s) = r'$ and $r \to r'$.*

Proof: By induction on c and case analysis on the first instruction and on S. The interesting case is $c = \texttt{ADD}; c'$.

If S is empty, we have $s = n.s'$ for some n and s', and $r = \mathcal{D}(\texttt{ADD}; c', a.n.s') = \mathcal{D}(c', (n + a).s')$. Note that $n + a \to n + a'$. By induction hypothesis, it follows that there exists r' such that $r \to r'$ and $\mathcal{D}(c', (n+a').s') = r'$. This is the desired result, since $\mathcal{D}(\texttt{ADD}; c', a'.n.s') = \mathcal{D}(c', (n+a').s') = r'$.

If $S = b.\varepsilon$ is empty, we have $r = \mathcal{D}(\texttt{ADD}; c', b.a.s) = \mathcal{D}(c', (a + b).s')$. Note that $a + b \to a' + b$. The result follows by induction hypothesis.

If $S = b_1.b_2.S'$, we have $r = \mathcal{D}(\texttt{ADD}; c', b_1.b_2.S'.a.s) = \mathcal{D}(c', (b_2 + b_1).S'.a.s')$. The result follows by induction hypothesis. □

Lemma 2 (Simulation) *If the HP calculator performs a transition from (c, s) to (c', s'), and $\mathcal{D}(c, s) = a$, there exists a' such that $a \xrightarrow{*} a'$ and $\mathcal{D}(c', s') = a'$.*

Proof: By case analysis on the transition.

Case CONST transition: $(\texttt{CONST}(N); c,\ s) \to (c,\ N.s)$. We have $\mathcal{D}(\texttt{CONST}(N); c,\ s) = \mathcal{D}(c,\ N.s)$ since the symbolic machine can perform the same transition. Therefore by definition of decompilation, the two states decompile to the same term. The result follows by taking $a' = a$.

Case ADD transition: $(\texttt{ADD}; c,\ n_2.n_1.s) \to (c,\ n.s)$ where the integer n is the sum of n_1 and n_2. We have $a = \mathcal{D}(\texttt{ADD}; c,\ n_2.n_1.s) = \mathcal{D}(c,\ b.s)$ where b is the expression $n_1 + n_2$. Since $b \to n$, the compatibility lemma therefore shows the existence of a' such that $a \to a'$ and $\mathcal{D}(c,\ n.s) = a'$. This is the desired result.

Case SUB transition: similar to the previous case. □

Lemma 3 (Initial state) *The state $(\mathcal{C}(a), \varepsilon)$ decompiles to a.*

Proof: We show by induction on a that the symbolic machine can perform transitions from $(\mathcal{C}(a).k, S)$ to $(k, a.S)$ for all codes k and symbolic stack S. (The proof is similar to that of theorem 10 in lecture II.) The result follows by taking $k = \varepsilon$ and $S = \varepsilon$. □

Lemma 4 (Final state) *If the machine stops on a state (c, s) that decompiles to the expression a, then (c, s) is a final state $(\varepsilon, n.\varepsilon)$ and $a = n$.*

Proof: By case analysis on the code c. If c is empty, by definition of decompilation we must have $s = n.\varepsilon$ and $a = n$ for some integer n. If c starts with a $\texttt{CONST}(N)$ instruction, the machine can perform a CONST transition and therefore is not stopped. If c starts with an ADD or SUB instruction, the stack s must contain at least two elements, otherwise the symbolic machine would get stuck and the decompilation of (c, s) would be undefined. Therefore, the concrete machine can perform an ADD or SUB transition and is not stopped. □

Exercise II.6 We show that for all n and a, if $a \Rightarrow \infty$, there exists a reduction sequence of length $\geq n$ starting from a. The proof is by induction over n and sub-induction over a. By hypothesis $a \Rightarrow \infty$, there are three cases to consider:

Case $a = b\ c$ and $b \Rightarrow \infty$. By induction hypothesis applied to n and b, we have a reduction sequence $b \xrightarrow{*} b'$ of length $\geq n$. Therefore, $a = b\ c \xrightarrow{*} b'\ c$ is a reduction sequence of length $\geq n$.

Case $a = b\ c$ and $b \Rightarrow v$ and $c \Rightarrow \infty$. By theorem 3 of lecture I, $b \xrightarrow{*} v$. By induction hypothesis applied to n and c, we have a reduction sequence $c \xrightarrow{*} c'$ of length $\geq n$. Therefore, $a = b\ c \xrightarrow{*} v\ c \xrightarrow{*} v\ c'$ is a reduction sequence of length $\geq n$.

Case $a = b\ c$ and $b \Rightarrow \lambda x.d$ and $c \Rightarrow v$ and $d[x \leftarrow v] \Rightarrow \infty$. By theorem 3 of lecture I, $a \xrightarrow{*} \lambda x.d$ and $b \xrightarrow{*} v$. By induction hypothesis applied to $n-1$ and $d[x \leftarrow v]$, we have a reduction sequence $d[x \leftarrow v] \xrightarrow{*} e$ of length $\geq n-1$. Therefore,

$$a = b\ c \xrightarrow{*} (\lambda x.d)\ c \xrightarrow{*} (\lambda x.d)\ v \rightarrow d[x \leftarrow v] \xrightarrow{*} e$$

is a reduction sequence of length $\geq 1 + (n-1) = n$.

Exercise II.7 For question (1), we show that $\forall a, \mathcal{E}_n(a) \leq \mathcal{E}_{n+1}(a)$ by induction over n. The base case $n = 0$ is obvious since $\mathcal{E}_0(a) = \perp$. For the inductive case, we assume the result for n and consider $\mathcal{E}_{n+1}(a)$ by case over a. The non-trivial case is $a = b\ c$. We know (induction hypothesis) that $\mathcal{E}_n(b) \leq \mathcal{E}_{n+1}(b)$ and $\mathcal{E}_n(c) \leq \mathcal{E}_{n+1}(c)$.

If $\mathcal{E}_n(b) = \perp$ or $\mathcal{E}_n(c) = \perp$, then $\mathcal{E}_{n+1}(a) = \perp$ and the result is obvious.

Otherwise, $\mathcal{E}_{n+1}(b) = \mathcal{E}_n(b)$ and $\mathcal{E}_{n+1}(c) = \mathcal{E}_n(c)$, from which it follows that either $\mathcal{E}_{n+2}(a) = \mathtt{err} = \mathcal{E}_{n+1}(a)$, or $\mathcal{E}_{n+2}(a) = \mathcal{E}_{n+1}(d[x \leftarrow v'])$ and $\mathcal{E}_{n+1}(a) = \mathcal{E}_n(d[x \leftarrow v'])$ for the same d and v', and the result follows by induction hypothesis.

We then conclude that $\mathcal{E}_n(a) \leq \mathcal{E}_m(a)$ if $n \leq m$ by induction on the difference $m - n$ and transitivity of $\leq$.

Consider now the sequence $(\mathcal{E}_n(a))_{n \in \mathbf{N}}$ for a fixed a. Either $\forall n, \mathcal{E}_n(a) = \perp$, or $\exists n, \mathcal{E}_n(a) \neq \perp$. In the first case, the sequence is constant and equal to $\perp$, hence $\lim_{n \to \infty} \mathcal{E}_n(a) = \perp$. In the second case, for all $m \geq n$, $\mathcal{E}_m(a) \geq \mathcal{E}_n(a) \neq \perp$, that is, $\mathcal{E}_m(a) = \mathcal{E}_n(a)$. The sequence is therefore constant starting from rank n, hence $\lim_{m \to \infty} \mathcal{E}_m(a)$ is defined and equal to $\mathcal{E}_n(a)$.

This limit corresponds to the behavior of the `eval` Caml function, in the following sense: if the limit is a value v, `eval` a terminates and returns v; if the limit is `err`, `eval` a terminates on an uncaught exception `Error`; and if the limit is $\perp$, `eval` a loops.

For question (2), we show that $a \Rightarrow v$ implies $\exists n, \mathcal{E}_n(a) = v$ by induction on the derivation of $a \Rightarrow v$. The cases $a = N$ and $a = \lambda x.b$ are trivial: take $n = 1$. For the case $a = b\ c$, the induction hypothesis gives us integers p, q, r such that

$$\mathcal{E}_p(b) = \lambda x.d \qquad \mathcal{E}_q(c) = v' \qquad \mathcal{E}_r(d[x \leftarrow v'] = v$$

Taking $n = 1 + \max(p, q, r)$ and using the monotonicity of $\mathcal{E}$, we have that $\mathcal{E}_n(b\ c) = v$.

Conversely, we show that $\mathcal{E}_n(a) = v$ implies $a \Rightarrow v$ by induction over n and case analysis over a. Again, the cases $a = N$ and $a = \lambda x.b$ are trivial: we must have $v = a$. For the case $a = b\ c$, the fact that $\mathcal{E}_{n+1}(a) = v$ (and not `err` neither $\perp$) implies that

$$\mathcal{E}_n(b) = \lambda x.d \qquad \mathcal{E}_n(c) = v' \qquad \mathcal{E}_n(d[x \leftarrow v'] = v$$

The result follows by induction hypothesis applied to these three computations, and an application of the (app) rule.

For question (3), we show $\forall a,\ a \Rightarrow \infty$ implies $\mathcal{E}_n(a) = \bot$ by induction over n. The case $n = 0$ is trivial. Assuming this property for n, we consider the evaluation rule that concludes $a \Rightarrow \infty$. For instance, if $a = b\ c$ and $b \Rightarrow \infty$, by induction hypothesis, $\mathcal{E}_n(b) = \bot$, from which it follows that $\mathcal{E}_{n+1}(a) = \bot$. The proof is similar for the other two rules.

For the converse implication ($\forall n,\ \mathcal{E}_n(a) = \bot$ implies $a \Rightarrow \infty$), see the paper *Coinductive big-step semantics*.

Part III: Program transformations

Exercise III.1 The translation rule for λ-abstraction needs to be changed:

$$
\begin{array}{rcl}
[\![\lambda x.a]\!] & = & \texttt{tuple}(\lambda c, x.\ \texttt{let}\ x_1 = \texttt{field}_1(c)\ \texttt{in} \\
& & \qquad\qquad \ldots \\
& & \qquad\qquad \texttt{let}\ x_n = \texttt{field}_n(c)\ \texttt{in} \\
& & \qquad\qquad [\![a]\!], \\
& & \qquad x_1, \ldots, x_n)
\end{array}
$$

so that the variables $x_1, \ldots, x_n$ are not just the free variables of $\lambda x.a$, but all variables currently in scope. To do this, the translation scheme should take the list of such variables as an additional argument V:

$$
\begin{array}{rcl}
[\![x]\!]_V & = & x \\
[\![\lambda x.a]\!]_V & = & \texttt{tuple}(\lambda c, x.\ \texttt{let}\ x_1 = \texttt{field}_1(c)\ \texttt{in} \\
& & \qquad\qquad \ldots \\
& & \qquad\qquad \texttt{let}\ x_n = \texttt{field}_n(c)\ \texttt{in} \\
& & \qquad\qquad [\![a]\!]_{x.V}, \\
& & \qquad x_1, \ldots, x_n) \\
& & \text{where } V = x_1 \ldots x_n \\
[\![a\ b]\!]_V & = & \texttt{let}\ c = [\![a]\!]_V\ \texttt{in}\ \texttt{field}_0(c)(c, [\![b]\!]_V) \\
[\![\texttt{let}\ x = a\ \texttt{in}\ b]\!]_V & = & \texttt{let}\ x = [\![a]\!]_V\ \texttt{in}\ [\![b]\!]_{x.V}
\end{array}
$$

Exercise III.2 For a two-argument function $\lambda x.\lambda x'.a$, the two-argument method `apply2` will be defined as `return` $[\![a]\!]$. The one-argument method `apply` will build an intermediate closure (corresponding to $\lambda x'.a$) which, when applied, will call back to `apply2`.

Symmetrically, for a one-argument function $\lambda x.a$, we define `apply` as `return` $[\![a]\!]$ and `apply2` as calling `apply` on the first argument, then applying again the result to the second argument.

We encapsulate this construction in the following generic classes, from which we will inherit later:

```
abstract class Closure {
  abstract Object apply(Object arg);
  Object apply2(Object arg1, Object arg2) {
    return ((Closure)(apply(arg1))).apply(arg2);
  }
}
abstract class Closure2 extends Closure {
  Object apply(Object arg) {
    return new PartialApplication(this, arg);
  }
  abstract Object apply2(Object arg1, Object arg2);
}
class PartialApplication extends Closure {
  Closure2 fn; Object arg1;
```

```
  PartialApplication(Closure2 fn, Object arg1) {
    this.fn = fn; this.arg1 = arg1;
  }
  Object apply(Object arg2) {
    return fn.apply2(arg1, arg2);
  }
}
```

Now, the class generated for a two-argument function $\lambda x.\lambda y.a$ of free variables $x_1, \dots, x_n$ is

```
class C_{λx.λy.a} extends Closure2 {
    Object x_1; ...; Object x_n;
    C_{λx.λy.a}(Object x_1, ..., Object x_n) {
        this.x_1 = x_1; ...; this.x_n = x_n;
    }
    Object apply2(Object x, Object y) { return ⟦a⟧; }
}
```

The class generated for a one-argument function $\lambda x.a$ of free variables $x_1, \dots, x_n$ is

```
class C_{λx.λy.a} extends Closure {
    Object x_1; ...; Object x_n;
    C_{λx.a}(Object x_1, ..., Object x_n) {
        this.x_1 = x_1; ...; this.x_n = x_n;
    }
    Object apply(Object x) { return ⟦a⟧; }
}
```

Finally, the translation of expressions receives one additional case for curried applications to two arguments:

$$[\![a\ b\ c]\!] \quad = \quad [\![a]\!].\texttt{apply2}([\![b]\!], [\![c]\!])$$

Exercise III.3 Quite simply,

$$[\![\texttt{match}\ a\ \texttt{with}\ x \to a \mid \texttt{exception}\ y \to c]\!] \quad = \quad \texttt{match}\ [\![a]\!]\ \texttt{with}\ V(x) \to [\![b]\!] \mid E(y) \to [\![c]\!]$$

Note that $\texttt{try}\ a\ \texttt{with}\ x \to b$ can then be viewed as syntactic sugar for

$$\texttt{match}\ a\ \texttt{with}\ y \to y \mid \texttt{exception}\ x \to b$$

Exercise III.4

$$N \,/\, s \Rightarrow N \,/\, s \qquad\qquad \lambda x.a \,/\, s \Rightarrow \lambda x.a \,/\, s$$

$$\frac{a \,/\, s \Rightarrow \lambda x.c \,/\, s_1 \quad b \,/\, s_1 \Rightarrow v' \,/\, s_2 \quad c[x \leftarrow v'] \,/\, s_2 \Rightarrow v \,/\, s'}{a\ b \,/\, s \Rightarrow v \,/\, s'} \qquad \frac{a \,/\, s \Rightarrow v \,/\, s'}{\texttt{ref}\ a \,/\, s \Rightarrow \ell \,/\, s' + \ell \mapsto v}$$

$$\frac{a \,/\, s \Rightarrow \ell \,/\, s'}{!a \,/\, s \Rightarrow s'(\ell) \,/\, s'} \qquad \frac{a \,/\, s \Rightarrow \ell \,/\, s_1 \quad b \,/\, s_1 \Rightarrow v \,/\, s'}{(a := b) \,/\, s \Rightarrow (\,) \,/\, s' + \ell \mapsto v}$$

Exercise III.5 After the assignment

```
fact := λn. if n = 0 then 1 else n * (!fact) (n-1)
```

the reference `fact` contains a function which, when applied to $n \neq 0$, will apply the current contents of `fact`, that is, itself, to $n - 1$. Therefore, the function `!fact` will compute the factorial of its argument.

More generally, a recursive function $\mu f.\lambda x.a$ can be encoded as

```
let f = ref (λx. Ω) in
f := (λx. a[f ←!f]);
!f
```

In an untyped setting, any expression Ω will do. In a typed language, Ω must have the same type as the function body a. A simple solution is to define Ω as an infinite loop (of type $\forall\alpha.\alpha$) or as `raise` of an exception (idem).

Programming exercise III.6 As suggested in the slides, we will use integers as unique identifiers for store locations, and applicative finite maps indexed by integers to capture the current contents of the store.

```
module IMap = Map.Make(struct type t = int let compare = Pervasives.compare end)
```

Assume given a type `value` for the values associated to integer locations by store contents. We will see later how to define this type. Then, the type of stores is:

```
type store = { nextkey: int; contents: value IMap.t }
```

and the empty store is

```
let store_empty = { nextkey = 0; contents = IMap.empty }
```

A location designating a value of type `'a` is a record of a unique integer (its key) and two functions to mediate between types `'a` and `value`:

```
type 'a location = {
  key: int;
  inj: 'a -> value;
  proj: value -> 'a
}
```

Reading and writing operations:

```
let store_read (l: 'a location) (s: store) : 'a =
  assert (l.key < s.nextkey);
  l.proj (IMap.find l.key s.contents)

let store_write (l: 'a location) (newval: 'a) (s: store) : store =
  assert (l.key < s.nextkey);
  { nextkey = s.nextkey;
    contents = IMap.add l.key (l.inj newval) s.contents }
```

Creating fresh locations is more delicate, because we need to invent the correct `inj` and `proj` morphism. This is tied to the actual definition of type `value`. To help forge intuitions, let's start by restricting ourselves to two types of stored values, `int` and `string`:

```
type value = Int of int | String of string
```

Allocating and initializing a location for an integer is, then,

```
let store_alloc_int (initval: int) (s: store) : int location * store =
  let l =
    { key = s.nextkey;
      inj = (function x -> Int x);
      proj = (function Int x -> x | _ -> assert false) } in
  (l, { nextkey = s.nextkey + 1;
        contents = IMap.add l.key (Int initval) s.contents })
```

Likewise, we can allocate and initialize locations for strings by replacing `int` by `string` and `Int` by `String` in the code above:

```
let store_alloc_string (initval: string) (s: store) : string location * store =
  let l =
    { key = s.nextkey;
      inj = (function x -> String x);
      proj = (function String x -> x | _ -> assert false) } in
  (l, { nextkey = s.nextkey + 1;
        contents = IMap.add l.key (String initval) s.contents })
```

But in general we want stored values of arbitrary types, not just `int` and `string`! To achieve this, we can use an *extensible datatype* for the type `value`. (Extensible datatypes were introduced in OCaml version 4.02.) That is, a type that starts with zero constructors, and can be extended dynamically with new constructors during program execution. The following declares the extensible datatype of values:

```
type value = ..
```

Now, we can write a `store_alloc` function that works for any type. It extends the type `value` with a new constructor `V` of the right type, then plays the same tricks as in `store_alloc_int` above, using the new constructor `V` instead of the fixed constructor `Int`.

```
let store_alloc (type a) (initval: a) (s: store) : a location * store =
  let module M = struct type value += V of a end in
  let l =
    { key = s.nextkey;
      inj = (function x -> M.V x);
      proj = (function M.V x -> x | _ -> assert false) } in
  (l, { nextkey = s.nextkey + 1;
        contents = IMap.add l.key (M.V initval) s.contents })
```

Even though all constructors of type `value` are named `V`, they have distinct identities: a fresh constructor, distinct from all previously allocated ones, is generated every time the `let module` above is evaluated.

In OCaml prior to version 4.02, there are no user-defined extensible datatypes. However, the type `exn` of exception values is a predefined extensible datatype. So, just take

```
type value = exn

let store_alloc (type a) (initval: a) (s: store) : a location * store =
  let module M = struct exception V of a end in
  ...
```

Exercise III.7

$$
\begin{aligned}
[\![a\ op\ b]\!] &= \lambda k.\ [\![a]\!]\ (\lambda v_a.\ [\![b]\!]\ (\lambda v_b.\ k(v_a\ op\ v_b)))\\
[\![C(a_1,\ldots,a_n)]\!] &= \lambda k.\ [\![a_1]\!]\ (\lambda v_1.\ \ldots [\![a_n]\!]\ (\lambda v_n.\ k(C(v_1,\ldots,v_n))))
\end{aligned}
$$

$$
\begin{aligned}
&[\![\texttt{match}\ a\ \texttt{with}\ C(x_1,\ldots,x_n) \to b \mid \ldots]\!]\\
&\quad = \lambda k.\ [\![a]\!]\ (\lambda v.\ \texttt{match}\ v\ \texttt{with}\ C(x_1,\ldots,x_n) \to [\![b]\!]\ k \mid \ldots
\end{aligned}
$$

Exercise III.8 We use a global reference to maintain a stack of continuations expecting exception values.

```
let exn_handlers = ref ([]: exn cont list)

let push_handler k =
  exn_handlers := k :: !exn_handlers

let pop_handler () =
  match !exn_handlers with
  | [] -> failwith "abort on uncaught exception"
  | k :: rem -> exn_handlers := rem; k
```

At any time, the top of this stack is the continuation that should be invoked to raise an exception.

```
let raise exn =
  throw (pop_handler ()) exn
```

Now, we should arrange that the continuation at the top of the exception stack always branches one way or another to the `with` part of the nearest `try...with`. We encode `try...with` as a call to a library function `trywith`:

$$
\begin{aligned}
[\![\texttt{raise}\ a]\!] &= \texttt{raise}\ a\\
[\![\texttt{try}\ a\ \texttt{with}\ x \to b]\!] &= \texttt{trywith}\ (\lambda\_.a)\ (\lambda x.b)
\end{aligned}
$$

The tricky part is the definition of the `trywith` function. In pseudo-code:

```
let trywith a b =
  push_handler <a continuation that evaluates b of its
                argument and returns from trywith>;
  let res = a () in
  pop_handler ();
  res
```

This way, if `a ()` evaluates without raising exceptions, we push a continuation that will never be called, compute `a ()`, pop the continuation and return the result of `a ()`. If `a ()` raises an exception e, the continuation will be popped and invoked, causing `b` e to be evaluated and its value returned as the result of the `trywith`.

The really tricky part is to capture the right continuation to push on the stack. The only way is to pretend we are going to apply `b` to some argument, and do a `callcc` in this argument:

```
b (callcc (fun k -> push_handler k; ...))
```

However, we do not want to evaluate this application of `b` if the continuation `k` is not thrown. We therefore use a second `callcc/throw` to jump over the application of `b` in the case where `a ()` terminates normally:

```
callcc (fun k1 -> b (callcc (fun k -> push_handler k; ...; throw k1 ...)))
```

We can now fill the `...`, obtaining:

```
let trywith a b =
  callcc (fun k1 ->
    b (callcc (fun k2 ->
         push_handler k2;
         let res = a () in
         pop_handler ();
         throw k1 res)))
```

Exercise III.9 Proceed by induction on a derivation of $a \Rightarrow v$ and case analysis on the last rule used. The base cases $a = N$ or $a = \lambda x.b$ are easy. For instance:

$$[\![N]\!]\ k = (\lambda k.\ k\ N)\ k \rightarrow k\ N = k\ [\![N]\!]_v$$

For an inductive case, consider $a = b\ c$ with $b \Rightarrow \lambda x.d$ and $c \Rightarrow v'$ and $d[x \leftarrow v'] \Rightarrow v$. We can build the following reduction sequence:

$$
\begin{array}{rl}
[\![a]\!]\ k = (\lambda k.\ [\![a]\!]\ (\lambda v_a.\ [\![b]\!]\ (\lambda v_b.\ v_a\ v_b\ k)))\ k & \\
\downarrow & (\beta_v \text{ reduction}) \\
[\![a]\!]\ (\lambda v_a.\ [\![b]\!]\ (\lambda v_b.\ v_a\ v_b\ k)) & \\
\downarrow + & (\text{induction hypothesis on 1st premise}) \\
(\lambda v_a.\ [\![b]\!]\ (\lambda v_b.\ v_a\ v_b\ k))\ [\![\lambda x.d]\!]_v & \\
\downarrow & (\beta_v \text{ reduction}) \\
[\![b]\!]\ (\lambda v_b.\ [\![\lambda x.d]\!]_v\ v_b\ k) & \\
\downarrow + & (\text{induction hypothesis on 2nd premise}) \\
(\lambda v_b.\ [\![\lambda x.d]\!]_v\ v_b\ k)\ [\![v']\!]_v & \\
\downarrow & (\beta_v \text{ reduction + definition of } [\![\cdots]\!]_v) \\
(\lambda x.\ [\![d]\!])\ [\![v']\!]_v\ k & \\
\downarrow & (\beta_v \text{ reduction + substitution lemma}) \\
[\![d[x \leftarrow v']]\!]\ k & \\
\downarrow + & (\text{induction hypothesis on 3rd premise}) \\
k\ [\![v]\!]_v &
\end{array}
$$

At the end of the proof, we used the substitution lemma $[\![a[x \leftarrow v]]\!] = [\![a]\!][x \leftarrow [\![v]\!]_v]$, which is easy to check for any value v by induction over a.

Exercise III.10 For the core constructs (variables, constants, abstractions, applications), the double-barreled translation basically ignores / propagates unchanged the second, exceptional continuation:

$$
\begin{array}{rcl}
[\![N]\!] & = & \lambda k_1.\lambda k_2.\ k_1\ N \\
[\![x]\!] & = & \lambda k_1.\lambda k_2.\ k_1\ x \\
[\![\lambda x.a]\!] & = & \lambda k_1.\lambda k_2.\ k_1\ (\lambda x.[\![a]\!]) \\
[\![a\ b]\!] & = & \lambda k_1.\lambda k_2.\ [\![a]\!]\ (\lambda v_a.\ [\![b]\!]\ (\lambda v_b.\ v_a\ v_b\ k_1\ k_2)\ k_2)\ k_2
\end{array}
$$

The `raise` a construct evaluates a and passes its value to the exception continuation k_2. Any exception arising during the evaluation of a should also go to k_2. The normal continuation k_1 is ignored.

$$[\![\texttt{raise}\ a]\!] = \lambda k_1.\lambda k_2.\ [\![a]\!]\ k_2\ k_2$$

For `try`...`with`, we evaluate the body with a new exceptional continuation that catches exceptions arising out of it:

$$[\![\texttt{try}\ a\ \texttt{with}\ x \to b]\!] = \lambda k_1.\lambda k_2.\ [\![a]\!]\ k_1\ (\lambda x.\ [\![b]\!]\ k_1\ k_2)$$

Exercise III.11 Local CPS-conversion gives:

```
let rec cps_map f l k =
  match l with
  | [] -> k []
  | hd :: tl ->
      let hd' = f hd in cps_map f tl (fun l' -> k (hd' :: l'))
let map f l = cps_map f l (fun l -> l)
```

We see two continuation forms that need defunctionalization:

```
type 'a funval =
| A                      (* fun l -> l *)
| B of 'a funval * 'a    (* fun l' -> k (hd' :: l')
```

As in the lecture, the type `'a funval` is isomorphic to `'a list`, so we will use `'a list` directly:

- the empty list `[]` encodes `fun l -> l`
- the cons list `hd' :: k` encodes `fun l' -> k (hd' :: l')`

With this representation, defunctionalization gives:

```
let rec defun_map f l k =
  match l with
  | [] -> apply k []
  | hd :: tl ->
      let hd' = f hd in defun_map f tl (hd' :: k)
and apply k l =
  match k with
  | [] -> []
  | hd' :: k' -> apply k' (hd' :: l)
let map f l = defun_map f l []
```

We recognize `apply` to be `rev_app` (reverse & append), obtaining:

```
let rec rev_app x y =
  match x with
  | [] -> y
  | x1 :: xs -> rev_app xs (x1 :: y)
let rec tail_map f l accu =
  match l with
  | [] -> rev_app accu []
  | hd :: tl -> tail_map f tl (f hd :: accu)
let map f l = tail_map f l []
```

Exercise III.12 The general shape of call-by-name CPS translated terms is similar to that of call-by-value CPS translated term: the translation of a expects a continuation and eventually applies it to a's value. Hence, if $a : \tau$, then

$$[\![a]\!] : ([\![\tau]\!] \rightarrow \texttt{answer}) \rightarrow \texttt{answer}$$

with $[\![\tau]\!] = \tau$ for base types, just like in call-by-value. The difference between call-by-value and call-by-name is apparent for function types. In call-by-value, a function of type $\tau_1 \rightarrow \tau_2$, after CPS translation, expects an argument that is already evaluated and therefore has type $[\![\tau_1]\!]$. In call-by-name, the function expects an argument that is not evaluated yet and therefore is expecting a continuation. The type of the argument is, therefore, $([\![\tau_1]\!] \rightarrow \texttt{answer}) \rightarrow \texttt{answer}$. Hence,

$$[\![\tau_1 \rightarrow \tau_2]\!] = (([\![\tau_1]\!] \rightarrow \texttt{answer}) \rightarrow \texttt{answer}) \rightarrow ([\![\tau_2]\!] \rightarrow \texttt{answer}) \rightarrow \texttt{answer}$$

A nicer presentation involves two type translations: $[\![\tau]\!]_u$ is the type for an unevaluated expression of type τ, and $[\![\tau]\!]_v$ (which we noted $[\![\tau]\!]$ before) is the type for an evaluated expression of type τ. Then, we have:

$$\begin{array}{rcl} [\![\tau]\!]_u & = & ([\![\tau]\!]_v \to \texttt{answer}) \to \texttt{answer} \\ [\![\texttt{int}]\!]_v & = & \texttt{int} \\ [\![\tau_1 \to \tau_2]\!]_v & = & [\![\tau_1]\!]_u \to [\![\tau_2]\!]_u \end{array}$$

Part IV: Monads

Exercise IV.1 The precise statement of the theorem we are going to prove is

Theorem 1 *If $a \Rightarrow r$ in the natural semantics for exceptions, then $[\![a]\!] \approx [\![r]\!]_r$, where $[\![r]\!]_r$ is defined by*

$$[\![v]\!]_r = \texttt{ret}\ [\![v]\!]_v \qquad [\![\texttt{raise}\ v]\!]_r = \texttt{raise}\ [\![v]\!]_v$$

The proof is by induction on a derivation of $a \Rightarrow r$ and case analysis on the last rule used. The cases where a is a core language construct that evaluates to a value v have already been proved in the generic proof given in the slides.

Case $(\texttt{try}\ b\ \texttt{with}\ x \to c) \Rightarrow v$ because $b \Rightarrow v$: by induction hypothesis, $[\![b]\!] \approx \texttt{ret}\ [\![v]\!]_v$. We have:

$$\begin{array}{rcl} [\![\texttt{try}\ a\ \texttt{with}\ x \to b]\!] & = & \texttt{trywith}\ [\![a]\!]\ (\lambda x.[\![b]\!]) \\ & \approx & \texttt{trywith}\ (\texttt{ret}\ [\![v]\!]_v)\ (\lambda x.[\![b]\!]) \\ & \approx & \texttt{ret}\ [\![v]\!]_v \end{array}$$

assuming that `trywith` satisfies hypotheses similar to those of `bind`, namely

5 $\texttt{trywith}\ (\texttt{ret}\ v)\ (\lambda x.b) \approx \texttt{ret}\ v$

6 $\texttt{trywith}\ a\ (\lambda x.b) \approx \texttt{trywith}\ a'\ (\lambda x.b)$ if $a \approx a'$

Case $(\texttt{try}\ b\ \texttt{with}\ x \to c) \Rightarrow r$ because $b \Rightarrow \texttt{raise}\ v$ and $c[x \leftarrow v] \Rightarrow r$. By induction hypothesis, $[\![b]\!] \approx \texttt{raise}\ v'$ and $[\![c[x \leftarrow v]]\!] \approx [\![r]\!]_r$.

$$\begin{array}{rcl} [\![\texttt{try}\ b\ \texttt{with}\ x \to c]\!] & = & \texttt{trywith}\ [\![b]\!]\ (\lambda x.[\![c]\!]) \\ & \approx & \texttt{trywith}\ (\texttt{raise}\ [\![v]\!]_v)\ (\lambda x.[\![c]\!]) \\ & \approx & [\![c]\!][x \leftarrow [\![v]\!]_v] \ = \ [\![c[x \leftarrow v]]\!] \\ & \approx & [\![r]\!]_r \end{array}$$

with one additional hypothesis:

7 $\texttt{trywith}\ (\texttt{raise}\ v)\ (\lambda x.b) \approx b[x \leftarrow v]$

Case $b\ c \Rightarrow \texttt{raise}\ v$ because $b \Rightarrow \texttt{raise}\ v$. By induction hypothesis, $[\![b]\!] \approx \texttt{raise}\ v'$.

$$\begin{array}{rcl} [\![b\ c]\!] & = & \texttt{bind}\ [\![b]\!]\ (\lambda v_b.\ldots) \\ & \approx & \texttt{bind}\ (\texttt{raise}\ [\![v]\!]_v)\ (\lambda v_b.\ldots) \\ & \approx & \texttt{raise}\ [\![v]\!]_v \end{array}$$

using the hypothesis

8 $\texttt{bind}\ (\texttt{raise}\ v)\ (\lambda x.b) \approx \texttt{raise}\ v$

Case $b\ c \Rightarrow$ `raise` v because $b \Rightarrow v'$ and $c \Rightarrow$ `raise` v.

$$\begin{array}{rcl} [\![b\ c]\!] & = & \texttt{bind}\ [\![b]\!]\ (\lambda v_b.\ \texttt{bind}\ [\![c]\!]\ (\lambda v_c \ldots)) \\ & \approx & \texttt{bind}\ (\texttt{ret}\ [\![v']\!]_v)\ (\lambda v_b.\ \texttt{bind}\ [\![c]\!]\ (\lambda v_c \ldots)) \\ & \approx & \texttt{bind}\ [\![c]\!]\ (\lambda v_c \ldots)) \\ & \approx & \texttt{bind}\ (\texttt{raise}\ [\![v]\!]_v)\ (\lambda v_c \ldots)) \\ & \approx & \texttt{raise}\ [\![v]\!]_v \end{array}$$

Other exception propagation rules are similar. It is easy to check hypotheses 5–8 by inspection of the definitions of `trywith` and `bind`.

Exercise IV.2 Just like the Logging monad is a simpler, more restricted form of State monad, the Diagnostics monad is a simpler, more restricted form of a State-and-Exception monad.

```
module Diag = struct
  type log = string list
  type 'a outcome = OK of 'a | Abort
  type 'a mon = log -> 'a outcome * log

  let ret (x: 'a) : 'a mon = fun l -> (OK x, l)
  let bind (m: 'a mon) (f: 'a -> 'b mon): 'b mon =
    fun l -> match m l with
             | (OK x, l') -> f x l'
             | (Abort, l') -> (Abort, l')

  type 'a result = 'a outcome * log
  let run (m: 'a mon): 'a result =
    match m [] with (x, l) -> (x, List.rev l)

  let log msg : unit mon = fun l -> (OK (), msg :: l)
  let abort : 'a mon = fun l -> (Abort, l)
end
```

Exercise IV.3 This is another simpler, more restricted form of the State monad. The current state of the name generator is a pair (n, m) where $0 \leq m < 26$ designates a letter among `a`…`z` and where n, if not zero, is the numerical suffix to add to the letter.

```
module NameSupply = struct
  type state = int * int
  type 'a mon = state -> 'a * state
  let ret (x: 'a) : 'a mon = fun s -> (x, s)
  let bind (m: 'a mon) (f: 'a -> 'b mon): 'b mon =
    fun s -> match m s with (x, s') -> f x s'
  let run (m: 'a mon): 'a =
```

```
    match m (0, 0) with (x, s) -> x

  let gensym: string mon =
    fun (n, m) ->
      let c = Char.chr (Char.code 'a' + n) in
      let name =
        if m = 0 then Printf.sprintf "%c" c else Printf.sprintf "%c%d" c m in
      let next =
        if n = 25 then (0, m+1) else (n+1, m) in
      (name, next)
end
```

Exercise IV.4

```
module ContAndException = struct
  type answer = int
  type α mon = (α -> answer) -> (exn -> answer) -> answer
  let return (x: α) : α mon = fun k1 k2 -> k1 x
  let bind (x: α mon) (f: α -> 'b mon) : 'b mon =
    fun k1 k2 -> x (fun vx -> f vx k1 k2) k2
  let raise exn : α mon =
    fun k1 k2 -> k2 exn
  let trywith (x : α mon) (f: exn -> α mon) : α mon =
    fun k1 k2 -> x k1 (fun e -> f e k1 k2)
  type α cont = α -> answer
  let callcc (f: α cont -> α mon) : α mon =
    fun k1 k2 -> f k1 k1 k2
  let throw (c: α cont) (x: α) : 'b mon =
    fun k1 k2 -> c x
  let run (c: answer mon) = c (fun x -> x) (fun _ -> failwith "uncaught exn")
end
```

Consider first the EC implementation using monad transformers. Expanding the type definitions, we have

$$\alpha\ \texttt{EC.mon} = (\alpha\ \texttt{outcome} \rightarrow \texttt{answer}) \rightarrow \texttt{answer}$$

In other words, in the EC combination, there is only one continuation, but it receives as arguments values of the α outcome type, that is, either $V(v)$ for normal results or $E(v)$ for exceptional results. Given such a continuation k, we can define the two continuations k_1, k_2 appropriate for the double-barreled approach as

$$k_1 = \lambda x.\ k\ (V(x)) \qquad k_2 = \lambda x.\ k\ (E(x))$$

Symmetrically, if we have two continuations k_1, k_2 of the double-barreled kind, we can reconstruct a single continuation for the EC approach:

$$k = \lambda x.\ \texttt{match}\ x\ \texttt{with}\ V(v) \rightarrow k_1\ v \mid E(e) \rightarrow k_2\ e$$

These two constructions outline an isomorphism between the terms produced by the double-barreled transformation and the alternate transformation. This isomorphism is an instance of the more general theory of *type isomorphisms* (see R. Di Cosmo's book), which says that

$$\alpha\ \texttt{outcome} \to \beta \approx (\alpha + \texttt{exn}) \to \beta \approx (\alpha \to \beta) \times (\texttt{exn} \to \beta)$$

and

$$\alpha \times \beta \to \gamma \approx \alpha \to \beta \to \gamma$$

are isomorphic types. Bottom line: modulo these isomorphisms, the `ContAndException` monad above is equivalent to the combination `EC = ExceptionTransf(Cont)`.

The `CE = ContTransf(Exception)` combination does not really work. Expanding the type definitions again, we have

$$\alpha\ \texttt{CE.mon} = (\alpha \to \texttt{answer outcome}) \to \texttt{answer outcome}$$

The final result of the program is indeed an `outcome` type, indicating whether the program terminates normally or on an uncaught exception. However, the continuations always receive a normal value of type α as argument: they cannot be passed an exception value, and therefore cannot handle an exception raised by a program fragment. This is apparent in the definition of `raise` using the `lift` operation of the `ContTransf` transformer:

```
let raise_CE (x: exn) : 'a CE.mon = CE.lift (Exception.raise x)
```

Expanding the definitions, we get:

```
let raise_CE (x: exn) : 'a CE.mon = fun k -> E x
```

That is, `raise x` immediately aborts the program, without executing the continuation.

Perhaps surprisingly, we can still define a `trywith` operation:

```
let trywith_CE (m: 'a CE.mon) (f: exn -> 'a CE.mon) : 'a CE.mon =
  fun k ->
    m (fun v -> match k v with V v -> V v | E e -> f e k)
```

This is the only definition that passes the type-checker. However, its behavior is not what we expect from an exception handler: the exceptions that are caught are not those raised by `m` (these still abort the program), but those raised by the continuation of `m`... If the continuation `k` applied to the value `v` of `m` raises an exception, `f` is called with this exception value and `k` as a continuation, causing the continuation to be executed a second time. This kind of backtracking on an exception is definitely not what we expect from a `try...with` construct, and it is doubtful that it has any practical usefulness.

Exercise IV.5 Let's consider lazy evaluation first. If we have a delayed function value of type $(\alpha \to \beta)$ `lazy`, and a delayed argument value of type α `lazy`, the canonical way to apply one to another while remaining lazy, is

```
let <*> f x = lazy ((force f) (force x))
```

Expressing this computation in terms of the `proj` and `cobind` operation of the Lazy comonad:

```
let <*> f x = cobind (fun f -> (proj f) (proj x)) f
```

Exercise IV.6 The teacher is still working on this one.