MPRI course 2-4-2
"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:

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

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 (it is a value that does not reduce), while a' reduces infinitely. The evaluation relation does not hold in these two cases.

Exercise I.4 The base case for the induction is $a = (\lambda x.c)\ v' \rightarrow c[x \leftarrow v'] = b$. 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 $a = a'\ c \rightarrow b'\ c = b$ where $a' \rightarrow b'$. 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 $a' \rightarrow b'$ 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 $a = v'\ a' \rightarrow v'\ b' = b$ where $a' \rightarrow b'$. 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 $a' \rightarrow b'$ 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.

Part II: Abstract machines

Exercise II.1

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

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 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 $a = a'[e']\ a_1[e_1]\ \ldots\ a_n[e_n]$. We now argue by case over a':

1. a' is a variable $\underline{m}$. Since a can reduce, it must be the case that $a' \xrightarrow{\varepsilon} e'(m)$, i.e. the m-th element of e' is defined. The machine code is $\mathcal{C}(a') = \texttt{ACCESS}(m)$. The machine can perform an `ACCESS` transition.

2. a' is an abstraction $\lambda.a''$. In this case, $n > 0$, otherwise a could not reduce. The code is $\mathcal{C}(a') = \texttt{GRAB}; \mathcal{C}(a'')$ and the stack is not empty, therefore the machine can perform a `GRAB` transition.

3. a' is an application $b\ c$. The code is $\mathcal{C}(a') = \texttt{PUSH}(\mathcal{C}(b)); \mathcal{C}(c)$. The machine can perform a `PUSH` transition.

Exercise II.3 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:

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

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 $a \rightarrow a'$. Then, there exists r' such that $\mathcal{D}(c, S.a'.s) = r'$ and $r \rightarrow r'$.*

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 $n + a \rightarrow n + a'$. By induction hypothesis, it follows that there exists r' such that $r \rightarrow r'$ 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 $a + b \rightarrow a' + b$. 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: $(\texttt{CONST}(N); c,\ s) \rightarrow (c,\ N.s)$. 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: $(\texttt{ADD}; c,\ n_2.n_1.s) \rightarrow (c,\ n.s)$ 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 $b \rightarrow n$, the compatibility lemma therefore shows the existence of a' such that $a \rightarrow a'$ and $\mathcal{D}(c,\ n.s) = a'$. This is the desired result.

Case `SUB` transition: similar to the previous case. □

Lemma 3 (Progress) *If $\mathcal{D}(c, s) = a$ and a can reduce, the machine can perform one transition from the state (c, s).*

Proof: By case 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, which contradicts the hypothesis that a reduces. If c starts with a `CONST(`N`)` instruction, the machine can perform a `CONST` transition. 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. □

Lemma 4 (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 5 (Final state) *The state $(\varepsilon, n.\varepsilon)$ decompiles to the expression n.*

Proof: Obvious by definition of decompilation. □

Exercise II.4 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,

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

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

Exercise II.5 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 $\mathcal{E}_0(a) = \bot$. For the inductive case, we assume the result for n and consider $\mathcal{E}_{n+2}(a)$ by case over a. The non-trivial case is $a = b\ c$. If $\mathcal{E}_n(b) = \bot$ or $\mathcal{E}_n(c) = \bot$, then $\mathcal{E}_{n+1}(a) = \bot$ 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 $\mathcal{E}_{n+2}(a) = \texttt{err} = \mathcal{E}_{n+1}(a)$, 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 $\forall n, \mathcal{E}_n(a) = \bot$, or $\exists n, \mathcal{E}_n(a) \neq \bot$. In the first case, the sequence is constant and equal to $\bot$, hence $\lim_{n\to\infty} \mathcal{E}_n(a) = \bot$. In the second case, for all $m \geq n$, $\mathcal{E}_m(a) \geq \mathcal{E}_n(a) \neq \bot$, 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 following Caml function applied to a:

```
let rec eval = function
  | Const n -> Const n
  | Var x -> raise Error
  | Lam(x, a) -> Lam(x, a)
  | App(a, b) ->
      let v = eval a in
      let v' = eval b in
      match v with Lam(x, c) -> eval (subst x v' c) | _ -> raise Error
```

That is, 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. The difference between this `eval` function and the one given in lecture I is that the former loops on terms such as $1\ \omega$ where ω diverges, while the latter raises `Error` in this case.

For question (2), we show $a \Rightarrow v \Rightarrow \exists n, \mathcal{E}_n(a) = v$ 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 $\mathcal{E}_n(a) = v \Rightarrow a \Rightarrow v$ 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` nor $\perp$) 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 $\forall a,\ a \Rightarrow \infty \Rightarrow \mathcal{E}_n(a) = \perp$ 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, $\mathcal{E}_n(b) = \perp$, from which it follows that $\mathcal{E}_{n+1}(a) = \perp$. The proof is similar for the other two rules.

The converse implication $(\forall n,\ \mathcal{E}_n(a) = \perp) \Rightarrow a \Rightarrow \infty$ follows from the coinduction principle applied to the set of terms

$$T = \{a \mid \mathcal{E}_n(a) = \perp \text{ for large enough } n\}$$

Consider a such that $\mathcal{E}_n(a) = \perp$ for large enough n. a must be an application $b\ c$, otherwise we would have $\mathcal{E}_n(a) \neq \perp$. There are only three possible cases:

- $\mathcal{E}_n(b) = \perp$ for large enough n;
- $\mathcal{E}_n(b) = v$ and $\mathcal{E}_n c = \perp$ for large enough n; all n;
- $\mathcal{E}_n(b) = \lambda x.d$ and $\mathcal{E}_n(c) = v$ and $\mathcal{E}_n(d[x \leftarrow v]) = \perp$ for large enough n.

Combined with question (2), this shows that T satisfies the hypothesis of the coinduction principle.

Part III: Program transformations

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

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

so that the variables $x_1, \dots, x_n$ 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:

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

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 $x_1, \ldots, x_n$ is

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

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

```
class C_{λx..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,

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

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

$$\texttt{lettry}\ y = a\ \texttt{in}\ y\ \texttt{with}\ x \to b$$

Exercise III.4

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

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

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

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).

Exercise III.6

$$
\begin{array}{rcl}
[\![a\ op\ b]\!] & = & \lambda k.\ [\![a]\!]\ (\lambda v_a.\ [\![b]\!]\ (\lambda v_b.\ k(v_a\ op\ v_b))) \\
[\![C(a_1,\ldots,a_n)]\!] & = & \lambda k.\ [\![a_1]\!]\ (\lambda v_1.\ \ldots [\![a_n]\!]\ (\lambda v_n.\ k(C(v_1,\ldots,v_n)))) \\
 & & [\![\texttt{match}\ a\ \texttt{with}\ C(x_1,\ldots,x_n) \rightarrow b \mid \ldots]\!] \\
 & = & \lambda k.\ [\![a]\!]\ (\lambda v.\ \texttt{match}\ v\ \texttt{with}\ C(x_1,\ldots,x_n) \rightarrow [\![b]\!]\ k \mid \ldots
\end{array}
$$

Exercise III.7 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`:

$$
\begin{array}{rcl}
[\![\texttt{raise}\ a]\!] & = & \texttt{raise}\ a \\
[\![\texttt{try}\ a\ \texttt{with}\ x \rightarrow b]\!] & = & \texttt{trywith}\ (\lambda z.a)\ (\lambda x.b)
\end{array}
$$

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 k2 -> push_handler k2; ...; 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)))
```

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 $\llbracket a \rrbracket \approx \llbracket r \rrbracket_r$, where $\llbracket r \rrbracket_r$ is defined by*

$$\llbracket v \rrbracket_r = \texttt{ret}\ \llbracket v \rrbracket_v \qquad \llbracket \texttt{raise}\ v \rrbracket_r = \texttt{raise}\ \llbracket v \rrbracket_v$$

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 $(\texttt{try}\ b\ \texttt{with}\ x \rightarrow c) \Rightarrow v$ because $b \Rightarrow v$: by induction hypothesis, $\llbracket b \rrbracket \approx \texttt{ret}\ \llbracket v \rrbracket_v$. We have:

$$\begin{array}{rcl} \llbracket \texttt{try}\ a\ \texttt{with}\ x \rightarrow b \rrbracket & = & \texttt{trywith}\ \llbracket a \rrbracket\ (\lambda x. \llbracket b \rrbracket) \\ & \approx & \texttt{trywith}\ (\texttt{ret}\ \llbracket v \rrbracket_v)\ (\lambda x. \llbracket b \rrbracket) \\ & \approx & \texttt{ret}\ \llbracket v \rrbracket_v \end{array}$$

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 $[\![c[x \leftarrow v]]\!] \approx [\![r]\!]_r$.

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

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, $[\![b]\!] \approx \texttt{raise}\ v'$.

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

using the hypothesis

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

Case $b\ c \Rightarrow \texttt{raise}\ v$ because $b \Rightarrow v'$ and $c \Rightarrow \texttt{raise}\ v$.

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

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

```
module ContAndException = struct
  type answer = int
  type α m = (α -> answer) -> (exn -> answer) -> answer
  let return (x: α) : α m = fun k1 k2 -> k1 x
  let bind (x: α m) (f: α -> β m) : β m =
    fun k1 k2 -> x (fun vx -> f vx k1 k2) k2
```

```
  let raise exn : α m =
    fun k1 k2 -> k2 exn
  let trywith (x : α m) (f: exn -> α m) : α m =
    fun k1 k2 -> x k1 (fun e -> f e k1 k2)
  type α cont = α -> answer
  let callcc (f: α cont -> α m) : α m =
    fun k1 k2 -> f k1 k1 k2
  let throw (c: α cont) (x: α) : β m =
    fun k1 k2 -> c x
  let run (c: answer m) = c (fun x -> x) (fun _ -> failwith "uncaught exn")
end
```

Exercise IV.3 The teacher is still working on this one.