MPRI course 2-4-2
"Functional programming languages"
Exercises

Xavier Leroy

Part I: Operational semantics

Exercise I.1 (*) Prove theorem 2 (the unique decomposition theorem). Hint: use the fact that a value cannot reduce.

Exercise I.2 (*) Check that the reduction rules for derived forms (`let`, `if/then/else`, `fst`, `snd`) are valid.

Exercise I.3 (*) Consider $a = 1\ 2$. Does there exist a value v such that $a \Rightarrow v$? Same question for $a' = (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$. Do you see anything different between these two examples?

Exercise I.4 (**) As claimed in the proof of theorem 4, show that if $a \rightarrow b$ and $b \Rightarrow v$, then $a \Rightarrow v$. Hint: proceed by induction on the derivation of $a \rightarrow b$ in SOS and by case analysis on the last rule used to derive $b \Rightarrow v$.

Part II: Abstract machines

Exercise II.1 (*) A Krivine machine is hidden in the ZAM. Can you find it? More precisely, define a compilation scheme $\mathcal{N}$ from λ-terms to ZAM instructions that implements call-by-name evaluation of the source term.

Exercise II.2 (**) Prove lemma 5 (the Progress lemma for Krivine's machine).

Exercise II.3 (***) State and prove the analogues of the Simulation, Progress, Initial state and Final state lemmas (lemmas 4–7 in the case of Krivine's machine) for the HP calculator. Use the notion of decompilation defined in the lecture notes and the following reduction semantics for arithmetic expressions:

$$N_1 + N_2 \rightarrow N \quad (\text{if } N = N_1 + N_2) \qquad N_1 - N_2 \rightarrow N \quad (\text{if } N = N_1 - N_2)$$

$$\frac{a \rightarrow a'}{a\ op\ b \rightarrow a'\ op\ b} \qquad \frac{b \rightarrow b'}{N\ op\ b \rightarrow N\ op\ b'}$$

Exercise II.4 (**) Complete the proof of theorem 14 (if $a \Rightarrow \infty$ then a reduces infinitely).

Exercise II.5 (***) Consider the following function $\mathcal{E}_n(a)$, defined by recursion over n, that evaluates a term a up to recursion depth n:

$$\begin{aligned}
\mathcal{E}_0(a) &= \bot \\
\mathcal{E}_{n+1}(x) &= \texttt{err} \\
\mathcal{E}_{n+1}(N) &= N \\
\mathcal{E}_{n+1}(\lambda x.a) &= \lambda x.a \\
\mathcal{E}_{n+1}(a\ b) &= \texttt{case}\ \mathcal{E}_n(a)\ \texttt{of}\ \bot \to \bot \mid \texttt{err} \to \texttt{err} \mid v \to \\
&\qquad \texttt{case}\ \mathcal{E}_n(b)\ \texttt{of}\ \bot \to \bot \mid \texttt{err} \to \texttt{err} \mid v' \to \\
&\qquad\quad \texttt{case}\ v\ \texttt{of}\ N \to \texttt{err} \mid\ \lambda x.c \to \mathcal{E}_n(c[x \leftarrow v'])
\end{aligned}$$

The result of $\mathcal{E}_n(a)$ is either a value v (denoting termination), the constant `err` (denoting an erroneous evaluation), or the constant $\bot$ (denoting an evaluation that cannot terminate at recursion depth n).

We define a partial order $\leq$ on evaluation results by $r \leq r$ and $\bot \leq r$ for all results r. Note that distinct values, as well as `err` and a value, are not comparable by $\leq$.

1. Show that $\mathcal{E}_n(a)$ is increasing in n, that is $\mathcal{E}_n(a) \leq \mathcal{E}_m(a)$ if $n \leq m$. Conclude that $\lim_{n\to\infty} \mathcal{E}_n(a)$ exists for all terms a. What is the relationship between this limit and the behavior of the Caml function `eval` defined in part I?
2. Show that $a \Rightarrow v$ if and only if $\exists n, \mathcal{E}_n(a) = v$.
3. Show that $a \Rightarrow \infty$ if and only if $\forall n, \mathcal{E}_n(a) = \bot$.

Part III: Program transformations

Exercise III.1 (*) How would you modify closure conversion so that it builds full closures rather than minimal closures?

Exercise III.2 (***) Consider again closure conversion targetting a class-based object-oriented language such as Java. (Slide 14.) How would you extend this transformation to efficiently handle curried applications to 2 arguments? Hint: each closure becomes an object with 2 methods, `apply` and `apply2`, performing applications to 1 and 2 arguments respectively.

Exercise III.3 (*) It has been proposed that Caml should be extended with a construct

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

that behaves not at all like `try` (`let` $x = a$ `in b`) `with` $y \to c$, but as follows:

$$\begin{aligned}
(\texttt{lettry}\ x = v\ \texttt{in}\ b\ \texttt{with}\ y \to c) &\xrightarrow{\varepsilon} b[x \leftarrow v] \\
(\texttt{lettry}\ x = \texttt{raise}\ v\ \texttt{in}\ b\ \texttt{with}\ y \to c) &\xrightarrow{\varepsilon} c[y \leftarrow v]
\end{aligned}$$

In other terms, the exception handler `with` $y \to c$ catches exceptions arising during the evaluation of a, but not those arising during evaluation of b. Extend the exception-returning conversion to deal with this `lettry` construct.

Exercise III.4 (**) Give a natural semantics for references. Hint: the evaluation predicate has the form $a/s \Rightarrow v/s'$ where s is the initial store at the beginning of evaluation and s' is the final store at the end of evaluation.

Exercise III.5 (**) What function is computed by the following expression?

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

Define a translation scheme from a functional language with recursive functions $\mu f.\lambda x.a$ to a functional language with only plain functions $\lambda x.a$ and references. Hint: a fixpoint combinator would do the job, but please use references instead.

Exercise III.6 (*) Define the CPS conversion of arithmetic operations a *op* b, constructor applications $C(a_1, \ldots, a_n)$ and pattern-matchings `match` a `with` $p_1 \mid \ldots \mid p_n$.

Exercise III.7 (***) Define a translation from exceptions to references + continuations. Hint: use an imperative stack containing continuations corresponding to the `with` part of active `try`...`with` constructs.

Part IV: Monads

Exercise IV.1 (**) Complete the proof of theorem 3 for a source language that includes exceptions (`raise` and `try`...`with`). To this end, you should prove that if $a \Rightarrow$ `raise` v, then $[\![a]\!] \approx$ `raise` $[\![v]\!]_v$. (The first `raise` corresponds to an exception result in the natural semantics for exception; the second `raise` is the corresponding operation of the exception monad.) What additional hypotheses do you need on the $\approx$ relation? Are they satisfied?

Exercise IV.2 (**) Implement (without using monad transformers) a monad that combines exceptions and continuations. Use the following representation for computations:

$$\texttt{type}\ \alpha\ \texttt{mon} = (\alpha \to \texttt{answer}) \to (\texttt{exn} \to \texttt{answer}) \to \texttt{answer}$$

That is, each computation takes *two* continuations as arguments: one to be called when the computation terminates normally, the other to be called when it terminates early because an exception is raised.

Exercise IV.3 (***) Extend the `Concur` monad transformer with synchronous communications over channels, in the style of CCS. For simplicity, channels will be identified by strings and the values exchanged over channels will be integers. Implement two additional operations

```
type channel = string
send: channel -> int -> unit mon
receive: channel -> int mon
```

A process doing `send` c n blocks until another process executes `receive` c for the same channel c. Then, both processes restart; the `send` returns `()` while the `receive` returns the integer n coming from the `send`.