MPRI course 2-4
"Functional programming languages"
Exercises

Xavier Leroy

October 13, 2016

Part I: Interpreters and 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 a difference between these two examples?

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

Programming exercise I.5 (**) Implement a naive interpreter that follows exactly the Wright-Felleisen presentation of CBV semantics. The core function to implement is `decomp`, taking a term a as argument and returning a pair of a reduction context E and a subterm a' such that $a = E[a']$ and a reduces iff a' reduces at head. How will you represent contexts? Compare the efficiency of this interpreter with that of the naive interpreter based on the SOS presentation shown in the course.

Exercise I.6 (*) We relax the (app-r) reduction rule as follows:

$$\frac{b \to b'}{a\ b \to a\ b'}\ \text{(app-r')}$$

That is, we allow reductions to take place on the right of an application, even if the left part of the application is not yet reduced to a value.

1. Show by way of an example that the reduction relation is no longer deterministic: give three terms a, a_1, a_2 such that $a \to a_1$ and $a \to a_2$ and $a_1 \neq a_2$.

2. Check that theorems 3 and 4 ($a \Rightarrow v$ if and only if $a \xrightarrow{*} v$) still hold for this relaxed, non-deterministic reduction relation. Conclude that the left-to-right evaluation order makes no difference for terminating terms.

3. Give an example of a term that evaluates differently under the two reduction strategies (left-to-right and non-deterministic).

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 (*) The $\mathcal{C}$ compilation schema for the ZAM is not mathematically precise: in the application case, it is not 100% clear what the k in $\texttt{PUSHRETADDR}(k)$ is, exactly. Reformulate the compilation schema more precisely as a 2-argument function $\mathcal{C}(a, k)$, which should prepend to the code k the instructions that evaluate the expression a and deposit its value on the top of the stack.

Exercise II.3 (/***)** How would you extend the ZAM and its compilation scheme to handle `if`/`then`/`else` conditional expressions?

Exercise II.4 (**) Prove lemma 8 (the Final states lemma for Krivine's machine).

Exercise II.5 (***) State and prove the analogues of the Simulation, Initial state and Final state lemmas (lemmas 6, 7, 8 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:

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

$$\frac{a \to a'}{a \; op \; b \to a' \; op \; b} \qquad\qquad \frac{b \to b'}{N \; op \; b \to N \; op \; b'}$$

Exercise II.6 (**) Complete the proof of theorem 14 (if $a \Rightarrow \infty$ then a reduces infinitely).

Exercise II.7 (***) Consider the following function $\mathcal{E}_n(a)$, defined by recursion over n, that evaluates a term a up to recursion depth n:

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

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 $\mathcal{E}_n(a) \leq \mathcal{E}_{n+1}(a)$. 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 $\forall n, \mathcal{E}_n(a) = \bot$. (The "if" part requires a proof by coinduction; do not attempt it if you are not familiar with the coinduction principle.)

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 (*) OCaml version 4.02 introduces support for the following construct

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

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

$$\begin{aligned}(\texttt{match}\ v\ \texttt{with}\ x \to a \mid \texttt{exception}\ y \to c) &\xrightarrow{\varepsilon} \quad b[x \leftarrow v] \\ (\texttt{match}\ (\texttt{raise}\ v)\ \texttt{with}\ x \to a \mid \texttt{exception}\ y \to c) &\xrightarrow{\varepsilon} \quad c[y \leftarrow v]\end{aligned}$$

In other words, 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 `match`-and-`try` 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.

Programming exercise III.6 (***) Write an OCaml implementation of the polymorphic store operations from slide 48 of lecture III. The implementation must be statically type-safe (no `Obj.magic`).

Exercise III.7 (*) 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.8 (***) 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.

Exercise III.9 (*) Prove the first case of Theorem 5: if $a \Rightarrow v$ and k is a value, then $[\![a]\!]\ k \xrightarrow{+} k\ [\![v]\!]_v$.

Exercise III.10 (**) — the "double-barreled" CPS transformation Define a CPS translation for a source language that includes mini-ML and exceptions (`raise`, `try`...`with`). Each source term a becomes a term $[\![a]\!] = \lambda k_1.\lambda k_2 \ldots$ that expects not one, but two continuations. The continuation k_1 is to be called on the value of a if a terminates normally. The continuation k_2 is to be called on an exception value if a terminates early by raising this exception.

Programming exercise III.11 (*) Using Danvy's et al's technique, derive a tail-recursive function equivalent to the familiar `map` function over lists:

```
let rec map f l =
    match l with [] -> [] | hd :: tl -> f hd :: map f tl
```

Exercise III.12 (**) Consider the CPS transformation for call-by-name given in the lecture. What is its effect on types? In other words, if a is a closed term of type τ, what is the type of its call-by-name CPS translation $[\![a]\!]$?

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 $a \Rightarrow \texttt{raise}\ \ v$, then $[\![a]\!] \approx \texttt{raise}\ [\![v]\!]_v$. (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 (*) Define a Diagnostics monad that, like the Logging monad, allow the programmer to log debug messages, but in addition provides an `abort` operation that stops the program when invoked. Make sure that the log is not erased when `abort` is called!

Exercise IV.3 (*) When implementing program transformations or other instances of symbolic processing, it is often necessary to generate fresh identifiers. Define a NameSupply monad that offers an operation `gensym` of type `string mon`. Every time `gensym` is used, it returns the next name from the sequence `a`, ..., `z`, `a1`, ..., `z1`, `a2`, ...

Exercise IV.4 (**) Implement (without using monad transformers) a monad that combines exceptions and continuations. Use the "double-barreled" approach from exercise III.9: computations should be represented as

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

(***) An alternate approach is to use monad transformers:

```
module EC = ExceptionTransf(Cont)
               (* Exception transformer applied to Contination monad *)
module CE = ContTransf(Exception)
               (* Continuation transformer applied to Exception monad *)
```

How do these alternate approaches compare with the "double-barreled" CPS transformation?

Exercise IV.5 (**) We saw that every monad is an applicative structure. Is this true for any comonad? How to define "comonadic application" for an arbitrary comonad? (Hint: use the lazy evaluation comonad as an example.)

Exercise IV.6 (****) Define an extended `Concur` monad transformer that features 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. The additional operations over channels are

```
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`.