Sémantique des Langages de Programmation (SemLP)
DM : Cooperative Concurrency

The language

We consider a simple extension to the monadic *call-by-value* λ-calculus of the lecture notes (Chapter 14) with :

1. Natural values,
2. Boolean values,
3. String values,
4. an *if-then-else* conditional expression,
5. references Chapter 17, and
6. two concurrency constructs : fork and yield.

The syntax of this extended language, which we shall call concurrent-λ, is given in Figure 1.[1] The symbol $\oplus$ represents a binary natural operator, $\otimes$ represents a natural binary comparator, and $\oslash$ represents a boolean binary operator. The special value $()$ is named *unit*, and it is the only value of type Unit.

$$
\begin{array}{lcll}
x & \in & \mathcal{V}ar & \text{(var. names)} \\
n & \in & \mathbb{N} & \text{(naturals)} \\
b & \in & \{\mathsf{true}, \mathsf{false}\} & \text{(booleans)} \\
s & \in & [\mathsf{a}-\mathsf{z}\mathsf{A}-\mathsf{Z}\mathsf{0}-\mathsf{9}]^* & \text{(strings)} \\
p & \in & \mathcal{R}ef & \text{(references)} \\
\\
V & ::= & \lambda x.\ M \mid n \mid b \mid s \mid p \mid () & \text{(values)} \\
\\
M & ::= & x \mid V \mid @(M, M) & \text{(terms)} \\
 & \mid & M \oplus M \mid -M \mid M \otimes M \mid M \oslash M \mid \neg M & \\
 & \mid & \mathsf{ref}\ M \mid !M \mid M := M & \\
 & \mid & \mathsf{let}\ x = M\ \mathsf{in}\ M & \\
 & \mid & \mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M & \\
 & \mid & \mathsf{fork}\ M \mid \mathsf{yield} & \text{(concurrency)} \\
 & \mid & \mathsf{print}\ M & \text{(print)} \\
\\
E & ::= & @(E, M) \mid @(V, E) & \text{(ev. contexts)} \\
 & \mid & E \oplus M \mid V \oplus E \mid -E \mid \ldots & \\
 & \mid & \mathsf{ref}\ E \mid !E \mid E := M \mid V := E & \\
 & \mid & \mathsf{let}\ x = E\ \mathsf{in}\ M \mid \mathsf{if}\ E\ \mathsf{then}\ M\ \mathsf{else}\ M & \\
 & \mid & \mathsf{print}\ E &
\end{array}
$$

FIGURE 1 – Syntax for concurrent-λ.

The semantics of most of the construct of this language are as given in the lecture notes. The print construct does is a side-effecting operation which shows a value on the terminal. The evaluation of a print term results in a unit "()".

1. You are free to **add** types and operators as you see fit.

Concurrency

The semantics of the fork M construct is to create a new thread of execution, returning a unit value "()", and continue executing until termination (or until the following yield). This means that the expression M can now be evaluated concurrently with the expression that created the new thread. However, unlike concurrent programming languages such as Java, the execution of a thread can only be voluntarily interrupted to give way for the execution of another thread. This is achieved by the execution of the yield command.
As an example consider the following program :

$$\mathsf{print}\ 0;\ \mathsf{fork}\ (\mathsf{print}\ 1);\ \mathsf{fork}\ (\mathsf{print}\ 2);\ \mathsf{yield};\ \mathsf{print}\ 3$$

can produce any of the results : 0123, 0321, 0231, etc. Where interpret sequence " ;" to be implemented as an application : $M; N = (\lambda x N)M$ where the variable x does not appear in N. We also assume that program termination has the effect of a yield. Otherwise we can assume that each fork is called with a command terminated by yield.
As can be seen from the example above, we assume that the choice of the thread to resume after a yield command is non-deterministic (including the yielding thread). However, a realistic implementation should strive to provide a fair scheduler that gives equal opportunity to all threads. Then for instance, under a Round Robin scheduler, we would expect the result 0123 for the program above.

Exercises

Exercice 1 : Implementing basic concurrent-λ

1. Give an operational semantics similar to that of Chapter 12 of the lecture notes to concurrent-λ. Notice that you will need the *heap* to give semantics to references [2], and you will need to record the set of threads currently executing.
2. Implement in your favorite programming language [3] an interpreter for concurrent-λ except for the concurrency constructs fork and yield. The input to the interpreter is an expresion in the *abstract syntax* of the language. You are not required to provide a parser. [4]
3. Implement the CPS transformation of Chapter 10 of the lecture notes for concurrent-λ.

Exercice 2 : Implementing concurrency

1. Implement the fork and yield constructs of concurrent-λ on top of the transformations given above. Provide a Round Robin scheduler for this implementation.
2. Provide a different scheduler of your choice and provide an example program where the Round Robin scheduler differs from the one provided.
3. Provide a test suite showing that your implementation of concurrent-λ is correct. In particular, consider different placements for yield commands to show changes in the behavior of programs.

2. The evaluation contexts are given. Identifying the redexes is left as part of the exercise.
3. OCaml ?
4. Providing a parser can be considered for extra credit.

Exercice 3 : Synchronization : Bonus

Consider adding an additional command wait M which blocks the execution of the current thread until the expression M evaluates to true. We assume that the expression M is effect-free. Whenever the expression does not evaluate to true, control is released as in the case of a yield command.

Submission

Submission Date : TBD
Submission Format : You have to submit the source code with appropriate :

1. README.md file explaining :
   - how to compile : should be completely automated,
   - how to run a set of predefined (by you) tests,
   - all the command line options, and if needed, the source syntax and format for your interpreter.
2. a builder file (Makefile, sbt, maven, ...),
3. the source code with reasonable comments,
4. a document providing the solutions to exercise 1.1, and a reasonable explanation of the tests you conducted with the expected results,
5. if your implementation does not immediately compile you might be requested to submit a virtual machine (or docker container) with a running implementation of your project.