Advanced Programming (I00032)
A Non-Deterministic Semantics

Assignment 11

1 Non-Determinism

Non-determinism means that a program (or a set of programs) does not have a single meaning, but that the actual meaning depends on non-controllable factors, which are different each time programs are executed. This situation is common for concurrent environments, in which a set of programs access the same memory location, file or database. The result depends on factors such as, the order in which execution of programs is scheduled, other processes, user input and network traffic.

Still the result of such concurrent programs is not arbitrary, but there is a set of possible outcomes. So a non-deterministic semantics assigns a set of possible results, instead of a single one.

2 Concurrent DSL

To keep things simple, we consider a very simplistic concurrent setting. There is only a single stored value, which is shared among processes, which we just assume to be an integer. To avoid runtime errors we assume it is 0 initially.

We also use a very simplistic deeply embedded DSL for manipulating this stored value:

```
:: Prog :== [Instr]

:: Instr = Write Expr

:: Expr = Int   Int
        | Plus  Expr Expr
        | Times Expr Expr
        | Read
```

A program consists of a sequence of instructions. The only instruction is to write the value of an expression to the store. Expressions can be integer constants, multiplication of expressions, addition of expression and finally the current value of the stored value. So there are no runtime type errors.

If we only execute a single program, the semantics is straightforward. Consider the program:

```
prog0 = [ Write (Int 12)
        , Write (Plus Read (Int 1))
        ]
```

The result, so the value of the store, is 13 after execution. Consider another program:

```
prog1 = [ Write (Plus Read (Times 2))
        ]
```

If we run both programs concurrently, there are multiple possible outcomes. In case first `prog1` is executed and then `prog0`, the result is 13 again. The other way around, in case first `prog0` is executed and then `prog1`, the result is 26. A third possibility is that the instruction of `prog1` is executed in between the instructions of `prog0`. Then the result is 25.

We assume that instructions are atomic. So `Write (Plus Read Read)` is equivalent to `Write (Times Read (Int 2))`. The stored value cannot be changed by another program between the two read operation in the same expression.

3 The semantics

Define a function assigning a semantics to a set of programs:

```
possibleResults :: [Prog] → [Int]
```

As discussed, the semantics is a set of possible results, so final states of the stored integer.

You have to step-wise execute programs, so you have to keep track of the work still to do, similar to the reduct in the iTasks semantics. However, as we have a deeply embedded DSL, the reduct does not have to be a function, but can just be a modified list of instructions.

You do not have to use a monad in this assignment, but of course you can.

4 Atomic Set of Instructions

We extend our DSL by a construct to indicate that a set of instructions is atomic, which means that no other programs can continue execution between those instructions:

```
:: Instr = Write  Expr
         | Atomic [Instr]
```

Extend you semantic function accordingly. Add a test that illustrates the effect of making a set of instructions atomic.

5 Semantics in Clean

What is that advantage of specifying a semantics in a functional language, in contrast to using ordinary mathematical notations? What are the disadvantages, if any? Answer the questions, in particular for the semantics in this assignment and in general. Can you image cases in which you may not want to specify semantics in a functional language?

Deadline

The deadline for this exercise is December 7, 13:30h.