Semantics of Programming Languages
Exercise Sheet 4

Exercise 4.1 Reflexive Transitive Closure

A binary relation is expressed by a predicate of type \( R :: 's \Rightarrow 's \Rightarrow \text{bool} \). Intuitively, \( R \ s \ t \) represents a single step from state \( s \) to state \( t \).

The reflexive, transitive closure \( R^* \) of \( R \) is the relation that contains a step \( R^* \ s \ t \), iff \( R \) can step from \( s \) to \( t \) in any number of steps (including zero steps).

Formalize the reflexive transitive closure as an inductive predicate:

\[
\text{inductive star :: } ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'a \Rightarrow \text{bool})
\]

When doing so, you have the choice to append or prepend a step. In any case, the following two lemmas should hold for your definition:

\textbf{lemma star\_prepend: } \[ [r \ x \ y; \ star \ r \ y \ z] \Rightarrow star \ r \ x \ z \]
\textbf{lemma star\_append: } \[ [star \ r \ x \ y; \ r \ y \ z] \Rightarrow star \ r \ x \ z \]

Now, formalize the star predicate again, this time the other way round:

\textbf{inductive star\' :: } \('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'a \Rightarrow \text{bool})\]

Prove the equivalence of your two formalizations:

\textbf{lemma } “star \ r \ x \ y = star\' \ r \ x \ y”

Exercise 4.2 A Structured Proof on Relations

We consider two binary predicates \( T \) and \( A \) and assume that \( T \) is total, \( A \) is antisymmetric and \( T \) is a subset of \( A \). Show with a structured, Isar-style proof that then \( A \) is also a subset of \( T \):

\textbf{assumes } \( \forall \ x \ y. \ T \ x \ y \lor T \ y \ x \)
\textbf{and } \( \forall \ x \ y. \ A \ x \ y \land A \ y \ x \longrightarrow x = y \)
\textbf{and } \( \forall \ x \ y. \ T \ x \ y \longrightarrow A \ x \ y \)
\textbf{shows } \( A \ x \ y \longrightarrow T \ x \ y \)
Exercise 4.3  Avoiding Stack Underflow

A *stack underflow* occurs when executing an instruction on a stack containing too few values – e.g., executing an *ADD* instruction on an stack of size less than two. A well-formed sequence of instructions (e.g., one generated by *comp*) should never cause a stack underflow.

In this exercise, you will define a semantics for the stack-machine that throws an exception if the program underflows the stack.

Modify the *exec1* and *exec* - functions, such that they return an option value, *None* indicating a stack-underflow.

```ocaml
fun exec1 :: “instr ⇒ state ⇒ stack ⇒ stack option”
fun exec :: “instr list ⇒ state ⇒ stack ⇒ stack option”
```

Now adjust the proof of theorem *exec_comp* to show that programs output by the compiler never underflow the stack:

```ocaml
theorem exec_comp: “exec (comp a) s stk = Some (aval a s ≠ stk)”
```

Homework 4.1  Palindromes

 Submission until Tuesday, November 22, 10:00am.

Formalize a definition of palindromes as an inductive predicate *palindrome* and prove:

```ocaml
lemma “palindrome xs ⇒ rev xs = xs”
```

A palindrome is a list that reads the same from the front and the back.

Homework 4.2  Compilation to Register Machine

 Submission until Tuesday, November 22, 10:00am.

In this exercise, you will define a compilation function from expressions to register machines and prove that the compilation is correct.

The registers in our simple register machines are natural numbers:

```ocaml
type synonym reg = nat
```

The instructions are:

- load an integer value in register 0 (“Load Immediate”)
- load the value of a variable (from the memory state) in register 0
- Store the value of register 0 in some other register
• add to register 0 the value of another register

datatype instr = LDI val | LD vname | MV reg | ADD reg

Recall that a memory state is a function from variable names to integers. A register state will be a function from registers to integers.

type synonym rstate = “reg ⇒ int”

Complete the following definition of the function for executing an instruction given a memory state s and a register state σ, the result being a register state.

fun exec :: “instr ⇒ state ⇒ rstate ⇒ rstate” where
“exec (ADD r1) s σ = σ (0 := σ r1 + σ 0)”

Next define the function executing a sequence of register-machine instructions, one at a time. We have already defined for you the case of empty list of instructions. You need to add the recursive case.

fun execs :: “instr list ⇒ state ⇒ rstate ⇒ rstate” where
“execs [] s σ = σ”

We are finally ready for the compilation function. Your task is to define a function cmp that takes an arithmetic expression a and produces a list of register-machine instructions whose execution in any memory state and register state should lead to a register state having in 0 the value of evaluating a in that memory state. In addition to the expression a, the compiler (cmp) will take as it’s second argument a variable r. The compiler is allowed to freely overwrite all registers with value r’ > r but should leave the registers with value 0 < r’ ≤ r untouched. Now the intended behavior of cmp is:

• cmp (N n) r loads immediate value n
• cmp (V x) r loads x (into register 0)
• cmp (Plus a1 a2) r first compiles a1 placing the result in register 0, moves the value from register 0 to some other allowed auxiliary register, then compiles a2, again placing the result in register 0, and finally adds the content of register 0 to that of the auxiliary register.

Finally, you need to prove the following correctness lemma, which states that our register-machine compiler is correct, in that executing the compiled instructions of an arithmetic expression yields (in register 0) the same result as evaluating the expression.

lemma cmpCorrect: “execs (cmp a r) s σ 0 = aval a s”

Hint: For proving correctness, you will need auxiliary lemmas stating that exec commutes with list concatenation and that the instructions produced by cmp a r do not alter registers below r.