Prof. Dr. A. Poetzsch-Heffter
M.Sc. Peter Zeller
Dipl.-Inf. C. Feller

University of Kaiserslautern
Department of Computer Science
Software Technology Group

Exercise Sheet 7: Specification and Verification with Higher-Order Logic (Summer Term 2014)

Please prepare the marked tasks for the exercise on Wednesday, June 18, 2014

Exercise 1 Big Step Semantics

Please download the file `Sheet7_bigstep.thy` from the homepage. It includes a definition of the syntax and semantics of a simple imperative while-language.

Consider the following program:

```
vi := 1;
vsum := 0;
WHILE vi <= vn DO
        vsum := vsum + vi;
        vi := vi + 1
END
```

You can find the abstract syntax tree of this program as the definition `prog` in the given theory file.

a) (Prepare!) Use the given big step semantics to show that the program terminates in a state where `vsum` is 6, when in the initial state `vn` was 3.

   Use the given elimination rules like `stepSemi` and `stepAsgn` for this proof.

b) We now want to proof a more general property of this program which is not fixed to `vn` being 3. As the first step towards this goal, prove that the execution of the loop body adds the value of `vi` to `vsum` and increments `vi` by 1.

c) As the next step, prove that the while-loop will add the sum of the numbers between `vi` and `vn` to `vsum`.

d) Now prove that the complete program calculates the sum of the numbers from 0 to `vn`.

Exercise 2 Extending the Big Step Semantics

In this exercise you should extend the language and the big step semantics from the Exercise 1 with some new language constructs. The file `Sheet7_bigstep_ext.thy` contains definitions for the programs below. Use those examples to check if your language extensions work as desired.

a) Write down the semantics rules required for exercises b) and c) on paper.

b) Extend the language with a construct for local variables. The scope of the local variables should already be part of the abstract syntax. For example the following program should result in a state where $x = 2$ and $y = 4$.

```
x := 1; y := 2;
VAR x := y + 2 IN
        y := x
END;
x := x + 1
```

c) Extend the language with a `LOOP` construct. It should take an expression and a command as the loop body. If the expression evaluates to n, the body should be executed n times. For example this program will result in a state where $x = 9$:

```
x := 3;
LOOP x BEGIN
        x := x + 2
END
```

d) (optional) Add procedures and procedure calls to the language. A program then consists of a list of procedures and one main block. Procedures should have a list of parameters, but no return value.

Here is an example of a program using procedures and local variables:

```
MAIN
        CALL fib [4]
END

PROCEDURE fib [x] BEGIN
        IF x<=1 THEN
                r := 1
        ELSE
                CALL fib [x + -2];
                VAR a := r IN
                        CALL FIB [x + -1];
                        r := r + a
                END
        END
END
```