COMS22201: Language Engineering

CWK2: Coursework 2 (25%)

Due: 01/05/2016, csxor@bristol.ac.uk

This coursework will draw together the work you have done on semantics with the work you have done on compilation. The aim is for you to develop a denotational and axiomatic semantics for the `While` programming language extended with read and write statements as in the previous assessed coursework CWK1; and for you to modify your earlier compiler to return an abstract syntax tree in the form of a Haskell term. Before starting this final coursework, you should refamiliarise yourself with the `While` language definition from CWK1:

`https://www.cs.bris.ac.uk/Teaching/Resources/COMS22201/language.html`

The coursework is split into three questions, each of which which will require you to submit one or more files into the CWK2 unit component in SAFE. This coursework counts towards 25% of your unit mark and the deadline for submission is midnight on Sunday 1st May, 2016.

For convenience, two stub files are provided: one that contains the Haskell type definitions and function signatures that you will need to implement for Question 1; and another (based on test7.w from CWK1) that contains the `While` source code used for the two examples in Question 1, and which you will need to annotate with axiomatic correctness proofs for Question 3:

`https://www.cs.bris.ac.uk/Teaching/Resources/COMS22201/cwk2.hs`

`https://www.cs.bris.ac.uk/Teaching/Resources/COMS22201/cwk2.w`

In addition to modifying these two files, you will also need to submit a shell script `cwk2.sh` along with any supporting files needed for Question 2.

Question 1: Denotational Semantics [35 marks]

In this first question you will build on the Haskell functions you wrote for the previous un-assessed assignments CWK2p1 and CWK2p2 in order to develop a denotational semantics for the extended `While` language. The key idea will be to introduce new type constructors to represent the different read and write statements; and to introduce a new type `IOState` which extends the notion of a program state with two additional lists representing the inputs and outputs of a program. As shown in the file `cwk2.hs` this is easily done as follows.

First, the following constructors are added to the `Stm` datatype in order to represent program statements involving read or write operations:

`Read` - for reading in the value of a variable;

`WriteA` - for writing out the value of an arithmetic expression;

`WriteB` - for writing out the value of a Boolean expression;

`WriteS` - for writing out a given string;

`WriteLn` - for writing out a string consisting of a newline character.

Second, the following type synonyms are added in order to represent the values that are read into or written out by the program:

`Input` - to denote the values read by a program;

`Output` - to denote the strings written by a program;

`IOState` - to denote the combined inputs, outputs and state of a program.

Part A) [7 marks]

Begin by downloading `cwk2.hs` and adding definitions for the following functions that you should previously have implemented as part of your solutions to CWK2p1 and CWk2p2 (or re-visit those courseworks for a description of the intended behaviour of these functions):

- `fv_aexp :: Aexp -> [Var]`
- `fv_bexp :: Bexp -> [Var]`
- `a_val :: Aexp -> State -> Z`
- `b_val :: Bexp -> State -> T`
- `cond :: (a->T, a->a, a->a) -> (a->a)`
- `fix :: (a -> a) -> a`
- `update :: State -> Z -> Var -> State`

Part B) [3 marks]

Now write a function `fv_stm::Stm -> [Var]` that uses `fv_aexp` and `fv_bexp` to return the set of free variables in a program statement (which, in this context, is just another way of asking you to return a list that contains exactly one occurrence of each variable which appears in a given program).

Part C) [3 marks]

Define a constant `fac::Stm` representing the following program:

```
write('Factorial calculator'); writeln;
write('Enter number: ');
read(x);
write('Factorial of '); write(x); write(' is ');
y := 1;
while !(x=1) do (
  y := y * x;
  x := x - 1
);
write(y);
writeln;
writeln;
```

Part D) [3 marks]

Define a constant `pow::Stm` representing the following program:

```
write('Exponential calculator'); writeln;
write('Enter base: ');
read(base);
if 1 <= base then (
  write('Enter exponent: ');
  read(exponent);
  num := 1;
  count := exponent;
  while 1 <= count do (
    num := num * base;
    count := count - 1
  );
  write(base); write(' raised to the power of '); write(exponent); write(' is ');
  write(num)
) else (
  write('Invalid base '); write(base)
);
writeln
```

Part E) [15 marks]

Write a function `s_ds::Stm -> IOState -> IOState` such that `s_ds p (i,o,s)` returns the result of semantically evaluating program `p` in state `s` with input list `i` and output list `o` (in other words, your function should return the inputs, outputs and state when the program terminates). Note that read and write statements should exhibit the following behaviour:

A `write` statement should convert its argument into a string (if it is not already a string) and append it onto the end of the output list. A `read` statement should take the first value from the input list and assign it to the given variable. In order for the output to more closely resemble what an interactive command line would look like, a `read` statement should also print the value it has just read onto the output after enclosing it within angle brackets `< >`. A `writeln` statement should just output a string consisting of a newline character.

Part F) [4 marks]

Write a function `eval::Stm -> IOState -> (Input, Output, [Var], [Num])` such that `eval p (i,o,s)` computes the result of evaluating program `p` in state `s` with input list `i` and output list `o`; and then returns the final input list and output list together with a list of the variables appearing in the program and their respective values in the final state. In this way you should verify that `eval fac ([5], [], undefined)` returns a tuple of the following four lists:

```
(  [],
   ["Factorial calculator","\n","Enter number: ","<5>",
    "Factorial of ","5"," is ","120","\n","\n"],
   ["x" , "y"],
   [ 1  , 120]  )
```

Q2: Abstract Syntax [30 marks]

In this second question you will build on the compiler you previously wrote for CWK1 in order to develop a parser that converts a `While` program into a Haskell term of type `Stm` for which you just wrote a denotational semantics.

If your code is based on the CWK1 skeleton compiler then you might consider adding another option `-hs` (in a similar vein to `-lex`, `-syn` and `-irt`) to output Haskell. But you are free to implement your compiler any way you wish.

You should submit all of the source files for your compiler along with the following two specifically named files:

1. a text file `readme.txt` containing instructions how to make your compiler;
2. a Bash script `cwk2.sh` that takes one argument denoting an input filename and uses your compiler to write a Haskell representation of the program in that file to `stdout`.

You are advised to test your compiler by using it to create a Haskell term `test0` for the program `test0.w` from CWK1 and verifying that the query `eval test0 ([1,2],[],const 0)` successfully terminates with an output stream that ends with a `"5"` followed by a `"yes"`.

Q3: Axiomatic Semantics [35 marks]

In this third question you'll use an extension of the axiomatic semantics to prove the partial correctness of the factorial and power programs from earlier. The key is to regard the lists of input values and output expressions as special variables, denoted *IN* and *OUT*, that can be used in pre- and post-conditions.

By viewing a statement of the form `read v` as equivalent to a composition `v:=`$head$`(IN); IN:=`$tail$`(IN)`, and by viewing a statement `write e` as equivalent to an assignment `OUT:=`$append$`(OUT,[e])`, and by viewing `writeln` as equivalent to `write '\n'`, we can easily derive the following axioms for read, write and writeln statements respectively:

$$\text{read}\ \frac{}{\{P_{[\,IN \mapsto tail(IN)\,]\,[\,\mathtt{v} \mapsto head(IN)\,]}\}\ \mathtt{read\ v}\ \{P\}}$$

$$\text{write}\ \frac{}{\{P_{[\,OUT \mapsto append(OUT,[\mathtt{e}])\,]}\}\ \mathtt{write\ e}\ \{P\}}$$

$$\text{writeln}\ \frac{}{\{P_{[\,OUT \mapsto append(OUT,[\mathtt{'\backslash n'}])\,]}\}\ \mathtt{writeln}\ \{P\}}$$

Note that, in addition to allowing the special list variables *IN* and *OUT* to appear in pre- and post-conditions, we also allow standard list notation and operators like *head*, *tail*, *append* to be used on them. Note also that while *IN* is a list of integers, *OUT* is a list of expressions that (differently from the denotational semantics) may be arithmetics or Booleans as well as strings. Recall also that strings in `While` are written using single quotes.

In this way, if `fac` denotes the factorial program from Question 1 Part C, then the following assertion states that, if the head of *IN* is initially n then, upon termination of the program, the third element from end of *OUT* will be the factorial of n.

$$\{head(IN) = n)\}\ \mathtt{fac}\ \{OUT = append(\_, [n!, \_, \_])\}$$

Part A) [15 marks]

Begin by downloading the stub file `cwk2.w` and provide a tableau-based partial correctness proof of the the above assertion by completing the annotation of the factorial code in the top half of the file.

Part B) [20 marks]

Now provide a tableau-based partial correctness proof of the exponent code in the bottom half of the file with respect to pre and post-conditions which assert that, if the first two elements of *IN* are both initially greater than or equal to 1 then, upon termination, the penultimate element of *OUT* will be the former value raised to the power of the latter.