University of Sussex
Informatics

Spring 2024

Limits of Computation

Assignment 1 (Deadline 7.03.2024, 4pm)

You need to submit this coursework electronically at the correct E-submission point. You must submit a zipped directory that contains three files with the exact names as described below:

- a *pdf* file `questions.pdf` containing the answers to Questions 1–3 . Please make sure that *all* answers for Questions 1–3 are in *this single document*. In case you use Word, please ensure that you convert your file into a pdf document before submission[1].
- a runnable `WHILE` source file `concat.while` that contains the answer to Question 4.
- a runnable `WHILE` source file `STEPn.while` that contains the answers to Question 5.
- Any other files in your directory will be ignored, so include no other programs or files. Note that both `WHILE`-programs, for Question 4 and 5, are only allowed to call macros that are published on our Canvas site (or your answer for Question 4) and both must run in `hwhile`. For marking your assignment it is essential that you follow the above rules.
- Please use a standard zip program to zip the directory. If you work on a Unix machine or a Mac use the (normally) pre-installed zip program. If you use a Windows machine, use WinZip or 7-Zip. Please do not use other compression programs or formats as this might give you 0 marks as the markers may be unable to unzip.
- Please do not write your name anywhere, but it is advisable to *include your candidate number as comment* in each submitted file.
- Please make sure you *check after submission* that you actually have submitted the correct zipped directory of files.

[1] In Word, this works usually by using the printing menu and then saving as pdf instead of sending to a printer.

YOU MUST WORK ON THE ASSIGNMENT ON YOUR OWN! The standard rules for collusion and plagiarism apply and any cases discovered will be reported and investigated.

Questions

1. A *list of lists of numbers* is a list such that all its elements are lists of numbers. Consider the following examples:

   - `[[3,5],[],[1,1,1]]` is a list of list of numbers;
   - `[[[3,5]]],[],[1]]` is not a list of list of numbers due to the first element;
   - `[2,[1]]` is not a list of list of numbers due to the first element, but note that the tree that encodes this list also encodes `[[0,0],[1]]` which *is* a list of list of numbers.

   Note that an empty list can always be considered a list of numbers and a list of lists of numbers.

   The (semantics of) language `WHILE` uses the binary tree data type $\mathbb{D}$. Consider the following elements in $\mathbb{D}$:

   (a) $\langle\langle \mathrm{nil}.\mathrm{nil}\rangle.\langle\langle \mathrm{nil}.\mathrm{nil}\rangle.\langle\langle \mathrm{nil}.\mathrm{nil}\rangle.\langle\langle \mathrm{nil}.\mathrm{nil}\rangle.\mathrm{nil}\rangle\rangle\rangle\rangle\rangle$

   (b) $\langle\langle \mathrm{nil}.\mathrm{nil}\rangle.\langle\langle\langle \mathrm{nil}.\mathrm{nil}\rangle.\mathrm{nil}\rangle.\langle\langle \mathrm{nil}.\mathrm{nil}\rangle.\langle\langle\langle \mathrm{nil}.\mathrm{nil}\rangle.\mathrm{nil}\rangle.\langle \mathrm{nil}.\mathrm{nil}\rangle\rangle\rangle\rangle\rangle\rangle$

   (c) $\langle\langle \mathrm{nil}.\mathrm{nil}\rangle.\langle\langle \mathrm{nil}.\mathrm{nil}\rangle.\langle\langle \mathrm{nil}.\mathrm{nil}\rangle.\langle\langle \mathrm{nil}.\langle \mathrm{nil}.\mathrm{nil}\rangle\rangle.\langle \mathrm{nil}.\mathrm{nil}\rangle\rangle\rangle\rangle\rangle\rangle$

   Answer for EACH of the given trees (a)–(c) above BOTH of the following questions:

   i Does it encode a *list of numbers*? If it does which is the corresponding list of numbers?

   ii Does it encode a *list of lists of numbers*? If it does which is the corresponding list of lists of numbers?

   Use our encoding of datatypes in $\mathbb{D}$ as explained in the lectures. [18 marks]

2. Describe what the following `WHILE`-program `p` computes for arbitrary input:

```
p read X {
    Stack := cons X nil;
    Y := nil;
    while Stack {
        D := hd Stack;
        Stack := tl Stack;
        if D {
              Stack := cons (hd D) (cons (tl D) Stack)
        }
        else
        { Y := cons nil Y }
  }
}
write Y
```

Do not narrate how the program executes step by step, but provide/describe the *function* it implements. Your description should explain the result for *any* input $d \in \mathbb{D}$, and thus should refer to the input tree. [20 marks]

3. Given the following WHILE-program as data d, write a source WHILE-program p such that $\ulcorner p \urcorner = d$.

```
[ 0
,
   [ [:=, 1, [cons, [quote, nil], [quote, nil]]]
   ,
       [ while, [var, 0]
       ,
           [ [:=, 2, [hd, [var, 0]]]
             ,
               [ if, [var, 2]
               ,
                   [ [:=, 0, [tl, [var, 0]]]
                   ]
               ,
                   [ [:=, 1, [quote, nil]]
                   , [:=, 0, [quote, nil]]
                   ]
               ]
           ]
       ]
   ]
, 1
]
```

Make sure your program has correct core WHILE-syntax. [18 marks]

4. The concatenation of lists (where A::R denotes a list with first element A and rest list R like in Haskell) is defined as follows:

$$\begin{array}{lcl} [] \circ M & = & M \\ (A :: R) \circ M & = & A :: (R \circ M) \end{array}$$

So, for instance $[1,2,3] \circ [4,5] = [1,2,3,4,5]$.

Write a *single* WHILE-program concat.while that implements concatenation $\circ$ above on lists (encoded in WHILE). The two list arguments must be wrapped in a single argument list. So ensure that

$$[\![\texttt{concat}]\!]^{\texttt{WHILE}}(\ulcorner[[1,2,3],[4,5]]\urcorner) = \ulcorner[1,2,3,4,5]\urcorner$$

Submit the source code in a file called concat.while. You may call macros but *only* WHILE-programs published on our Canvas site (and please don't include those in your submission). You can use extended WHILE-syntax but your program must run correctly in hwhile. [18 marks]

5. Extend our WHILE-interpreter u.while written in WHILE (and available from our webpage with all necessary macros) to cover the following new features:

   (a) add a concatenation operator for lists as expression, i.e. extend the expression syntax as follows:

   $$\langle \textit{expression} \rangle ::= \ldots \mid \texttt{concat}\ \langle \textit{expression} \rangle\ \langle \textit{expression} \rangle$$

   So e.g. concat hd X cons nil Y is now a legal expression.

   (b) add a break statement to the choice of commands, i.e. extend the command syntax as follows:

   $$\langle \textit{command} \rangle ::= \ldots \mid \texttt{break}$$

   So e.g. the following is now a legal block { X:= hd Y; break }.

   The interpreter should evaluate concat E F by first evaluating E to e and F to f (the order of evaluation is not important as long as you build the result correctly) and then returning $e \circ f$ as result.

   The interpreter should evaluate break by immediately terminating the execution of the innermost while loop that contains break. If there is no such while loop body then break should terminate the entire program (returning what is the current value of the result variable). For instance, executing

```
p read Z {
   R := true;
   while Z {
      Y := hd Z;
      Z := tl Z;
      if Y {} else { R := false; break; R := true }
     }
 } write R
```

with input $\ulcorner[1,1,0,1,1]\urcorner$ will stop executing after the while loop body has been executed three times (as `Y` will be nil then), due to the `break` statement. The output of the program in this case will be $\ulcorner false\urcorner$. This is an efficient implementation of checking whether all elements of a list are 'true'. Consider now program `q`:

```
q read L {
   while L {
    R := true;
    Z := hd L;
    while Z {
      Y := hd Z;
      Z := tl Z;
      if Y {} else { R := false; break }
     };
     L := tl L;
     M := cons R M
    }
 } write M
```

If we run `q` with input $\ulcorner[[1,1,0,1,1],[1,1]]\urcorner$, executing the `break` command will terminate only the innermost loop, it won't affect the outer while loop, and the output in this case would be $\ulcorner[\texttt{true},\texttt{false}]\urcorner$.

Finally, program

```
r read L {
   Y := cons nil nil;
   break;
   Y := nil
 }
 write Y
```

with any input will always return ⌜1⌝, as `break` will immediately terminate execution.

For extending the self-interpreter (in `hwhile`), you will need to represent programs in this extended language as data, so you will need atoms for `@concat` and `@break` in `hwhile`. But `hwhile` does not have syntax sugar for user-defined additional atoms, so you must use number `4` for `@concat` and number `6` for `@break`. Don't change that as otherwise we cannot test your program. So, for instance,

```
[4, [quote, nil], [quote, nil]]
```

is representing `concat nil nil` as data. And

```
[6]
```

is representing command `break` as data. Note that no arguments need to be encoded for `break`, so the encoding uses only the operation's code.

Please be advised that in `hwhile` you are unable to write `@4`, instead simply use `4` and match against that in the switch statement of the step macro (`STEPn.while`). Note also that this macro is the only program you need to change to implement the above extensions. For any additional atoms you need to introduce to encode auxiliary atoms, use even numbers starting from 8, 10, 12, and so on.

Submit only the file `STEPn.while` as *source* code, i.e. a `WHILE` program. Any macro you call should be either `concat` (which you already included in the submission for the previous question) or one of the macros published on Canvas.

Test your interpreter but don't submit the test data. We will also use testing for marking, so please make sure your program runs without syntax error in as many cases as possible. Ensure that your interpreter runs at least for simple programs with extensions and still runs correctly for all core `WHILE` programs. But note that for just submitting my original step macro without any extensions you will receive 0 marks. Add comments where useful to help the marker understand what you're doing. This may be important if your code is not working correctly.

If your program contains *obvious syntax errors*, it may receive 0 marks, so please run your programs! [26 marks]