University of Sussex
Informatics

Spring 2021

Limits of Computation

Assignment 1 (Deadline 4.03.2021, 4pm)

You need to submit this coursework electronically at the correct E-submission point. You must submit a zipped directory that is named after your candidate number (e.g. 123456.zip) and contains three files with the exact names as described below:

1. a *pdf* file `questions.pdf` containing the answers to Questions 1–4 . Please make sure that *all* answers for Questions 1–4 are in *this single document*. In case you use Word, please ensure that you convert your file into a pdf document before submission[1].
2. a (ASCII, i.e. simple text) file called `progasdata.txt` that contains the answer to Question 5.
3. a runnable WHILE file `STEP1.while` that contains the answer to Questions 6.
4. Any other files in your directory will be ignored, so include no other programs or files. 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 document. Please make sure you *check after submission* that you actually have submitted the correct zipped directory of files.

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.

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

WHILE++ – Extending WHILE with an Iterator

For the following questions, we enrich our WHILE-language with a list-iterator (such containers you will know from Java where there are iterators for collection types). The extended language shall be called WHILE++.

The syntax of our list iterator statements is defined below adding a case to the command case of the grammar for the WHILE-language from Lecture 3, Slide 17.

$\langle command \rangle$	$::=$	$\ldots$ |	/* WHILE-commands */
		`foreach` $\langle var \rangle$ `in` $\langle expression \rangle \langle block \rangle$	/* list iterator */

The semantics of the iterator command `foreach X in E B` is as follows: first evaluate `E` to a list (recall that this is always possible). For each element d of this list execute block `B` assuming that variable `X` (occurring in `B`) has value d. The iteration works from the front of the list towards the end of the list. If the list is empty the block `B` is never executed and the iterator does nothing. Note that `E` is evaluated only once at the beginning, it is never re-evaluated, even if it contains any variable that is changed by `B`! After execution, the variable `X` contains the last element of the evaluated list `E` unless it is assigned a value in `B`. In the latter case it will retain the last value assigned to it.

Let us define store $\sigma = \{X{:}\text{nil}, Y{:}\ulcorner[0,1,2]\urcorner, Z{:}\text{nil}\}$. Here are some examples of the semantics of the iterator:

`foreach X in Y {Z:= cons X Z}` $\vdash \sigma \rightarrow \{X{:}\ulcorner 2\urcorner, Y{:}\ulcorner[0,1,2]\urcorner, Z{:}\ulcorner[2,1,0]\urcorner\}$
`foreach X in hd Z {Z:= cons X Z}` $\vdash \sigma \rightarrow \sigma$
`foreach X in Y {Z:= cons nil Z}` $\vdash \sigma \rightarrow \{X{:}\ulcorner 2\urcorner, Y{:}\ulcorner[0,1,2]\urcorner, Z{:}\ulcorner 3\urcorner\}$
`foreach X in Y {X:= cons nil X;Z:= cons X Z}` $\vdash \sigma \rightarrow \{X{:}\ulcorner 3\urcorner, Y{:}\ulcorner[0,1,2]\urcorner, Z : \ulcorner[3,2,1]\urcorner\}$
`foreach X in Y {Z:= cons X Z;Y:= tl Y}` $\vdash \sigma \rightarrow \{X{:}\ulcorner 2\urcorner, Y{:}\text{nil}, Z{:}\ulcorner[2,1,0]\urcorner\}$

We also extend "programs-as-data" to include this form of iterator. The corresponding rule for the encoding (see the encodings given in Lecture 6 on Slide 18) looks as follows

$$\ulcorner \texttt{foreach X in E B} \urcorner = [\texttt{for}, \mathit{varnum}_{\texttt{X}}, \ulcorner\texttt{E}\urcorner, \ulcorner\texttt{B}\urcorner]$$

where `for` is a new atom. In order to be able to deal with this in `hwhile` (our WHILE interpreter) that does not recognise `@for`, let us fix the encoding of `for` to be number `4`. So in program as data representation of WHILE++-programs you *must* use `4` to represent the foreach construct, e.g.

```
    foreach X in Y {Z:= cons X Z}
```

is represented as `[4,0,[var,1],[[:=,2,[cons,[var,0],[var,2]]]]`.
Finally, here is a simple sample program in `WHILE++`:

```
sample read L {
  X := hd L;
  Y := hd tl L;
  foreach A in X {
      if hd Y { Z:= cons A Z };
      Y:= tl Y;
  }
}
write Z
```

Below you fill find <u>SIX</u> questions and you must answer all of them. They cover the material of Lectures 1 to 7.

1. `WHILEL` uses the same datatype of binary trees as `WHILE` does. Consider now the following trees in $\mathbb{D}$:

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

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

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

   (d) $\langle\langle \text{nil.nil}\rangle.\langle\langle \text{nil.nil}\rangle.\langle\langle \text{nil}.\langle \text{nil.nil}\rangle\rangle.\langle \text{nil.nil}\rangle\rangle\rangle\rangle$

   According to our encoding of datatypes in $\mathbb{D}$, decide for each tree (a)–(d) whether it encodes

   i a *list of numbers*; if it does give the corresponding list.

   ii a *list of lists of numbers*; if it does give the corresponding list.

   Note that an empty list can always be considered a list of numbers and a list of lists of numbers. [20 marks]

2. Let `prog` be the `WHILE++`-program in Figure 1. Explain what program `prog` in Figure 1 returns as output for a given input $d \in \mathbb{D}$. In other words, state what $[\![\texttt{prog}]\!]^{\texttt{WHILE++}}(d)$ is for any $d \in \mathbb{D}$. (Don't describe the code in the program.) [14 marks]

```
prog read L {
 foreach X in L
 {
  while X {
    Y:= cons nil Y;
    X:= tl X
    }
 }
}
write Y
```

Figure 1: `WHILE++`-program `prog`

3. The list iterator of `WHILE++`, as introduced above, could also be regarded as a mere syntactic extension of `WHILE` (i.e. "syntax sugar"). To demonstrate this, explain in detail how the command

   ```
   foreach X in E {S}
   ```

   can be translated into `WHILE` (without list iterators obviously) maintaining its original behaviour (i.e. semantics). Consider all possible forms of appearances of the iterator. Explain any assumptions you make. [18 marks]

4. According to the above, can we decide more problems with `WHILE++` programs than we can decide with `WHILE`-programs? Explain your answer briefly. [10 marks]

5. Translate the program `prog` in Figure 1 into program-as-data notation so that you can use it to test the answer for the next question. Submit this as a text file named `progasdata.txt`. This file must only contain text (no Word documents here!) representing the program as `WHILE++` data. Make sure you follow the specification of `WHILE++` programs-as-data format as given further above. [16 marks]

6. We would like to extend the self-interpreter `u.while` for `WHILE` (discussed in Lecture 7 and available from our Canvas site) so that it can also interpret `WHILE++` programs in abstract syntax. Due to the architecture of the self-interpreter program you *only need to change* the implementation of the step macro `STEPn.while` (which will be released on our Canvas site just after the last seminar at the end of Week 4). Note that you just have to add the

required additional code for the iterator. Submit the answer to this question in a separate text file `STEPL.while`.

One must be able to successfully run the universal program `u` as interpreter for `WHILE++` changing its `STEPn.while` macro call to a `STEPL.while` macro call. This is how we will test your answer and you should test yours in this way before submitting. You won't get marks if your `STEPL` macro is syntactically incorrect. Also make sure you test that your code does not lead to non-termination. This often happens when one does not clear the command stack properly.

In your `STEPL.while` program you may use macro calls to any program *published on the Canvas site* but if you do so, please don't include them in your submission. You must not call any self-defined programs. Add some comments to the code you add to help the marker understand what you are doing. [22 marks]