University of Sussex
Informatics

Spring 2020

Limits of Computation

Assignment 1 (Deadline 5.03.2020, 4pm)

You need to submit your answers electronically on Canvas at the E-submission point. You must submit several files as specified in this assignment. One file must be a *pdf* file containing the answers to Questions 1–4. Please make sure that all answers for Questions 1–4 are in *one single document*. In case you use Word, please ensure that you convert your file into a pdf document before submission[1].

Your submission must also contain the files containing the programs that answer Questions 5 and 6. For Question 5 you must submit a file `concat.while` and a file `concat.F`. For Question 6 you submit a file `intF.while` and potentially additional programs that `intF` calls as macros. For our test rig to be able to execute your files, please give the files the specified name. You may call programs published on the Canvas site and those don't need to be included in your submission.

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 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.

There are SIX questions and you must answer all of them. They cover the material of Lectures 1 to 7.

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

The language F

The language F is a simple functional language that uses the data type of binary trees you know from WHILE. F-programs consist of one expression that can make use of one recursive function definition.

Let us make this more precise by giving the syntax of the language.

$$
\begin{array}{lcl}
\langle \textit{program} \rangle & ::= & \texttt{in X out } \langle \textit{expression} \rangle \texttt{ where f(X) = } \langle \textit{expression} \rangle \\
\langle \textit{expression} \rangle & ::= & \texttt{X} \\
 & & \texttt{nil} \mid \\
 & & \texttt{cons } \langle \textit{expression} \rangle \, \langle \textit{expression} \rangle \mid \\
 & & \texttt{hd } \langle \textit{expression} \rangle \mid \\
 & & \texttt{tl } \langle \textit{expression} \rangle \mid \\
 & & \texttt{if } \langle \textit{expression} \rangle \texttt{ then } \langle \textit{expression} \rangle \texttt{ else } \langle \textit{expression} \rangle \mid \\
 & & \texttt{f(}\langle \textit{expression} \rangle\texttt{)}
\end{array}
$$

A program is an expression with *one* (fixed) input variable X and one output which is the result of an expression. The expression can refer to *one* recursive definition of a function called f with fixed formal parameter X. Note that X is the only variable in use, as (global) input variable to the program as well as the local argument of function f. The latter means that f can never directly access the program's global (input) variable.

The first five clauses for expressions are also available in (*core*) WHILE. But an F-expression can also be a conditional[2] or a function application of the form f(E). Note that there is always just one such function definition and the name of the function is *fixed* as f. The function may, of course, be called many times, also in a nested fashion. For instance, cons f(X) f(nil) is a legal expression as are f(f(X)) and f(cons f(hd X) f(f(X))).
As an example, here is the WHILE-program add from our lectures written as F-program. As usual, the two inputs are wrapped up in a single list:

```
in X out f(X)  where  f(X) =
     if   hd X
     then cons nil f(cons tl hd X  cons hd tl X  nil)
     else hd tl X
```

It implements the usual recursive definition of addition :

$$
\begin{array}{lcll}
n+m & = & \mathrm{succ}\,(\mathrm{pred}\,n+m) & \text{if } n>0 \\
n+m & = & m & \text{if } n=0
\end{array}
$$

[2] In WHILE we also have these conditionals but as statements.

where succ n denotes the successor of n (which in binary tree encoding terms is `cons nil` $\ulcorner n \urcorner$) and pred n denotes the predecessor of n (which in binary tree encoding terms is `tl` $\ulcorner n \urcorner$). Note that the input is a list of two numbers so `hd X` and `hd tl X` correspond to n and m, respectively.

The semantics of `F`-programs is given by a function

$$[\![\bullet]\!]^{\mathtt{F}} : \mathtt{F}\text{-Program} \to \mathbb{D} \to \mathbb{D}_\perp$$

defined formally as follows:

$$[\![\mathtt{in\ X\ out}\ E\ \mathtt{where\ f(X)} = B]\!]^{\mathtt{F}}\ v = \mathcal{F}[\![E]\!]\ B\ v$$

where the expression semantics function $\mathcal{F}[\![\bullet]\!]$ has type:

$$\mathcal{F}[\![\bullet]\!] : \mathtt{F}\text{-expression} \to \mathtt{F}\text{-expression} \to \mathbb{D} \to \mathbb{D}_\perp$$

First of all, compared to the expression semantics of `WHILE`, we note that there is an extra second argument of type `F`-expression, B. This is used to "carry around" (or "memorise" , "pass along") the body of the function definition such that it can be used whenever we find a call to function `f`. We also note that there is no store argument. This is obvious considering that the language `F` is a functional language so there is no assignment and thus there are no variables to assign to. But there is variable `X` which receives its value from the program call (used as input variable) or as formal parameter of function `f` (if used as formal parameter of `f`). The value of `X` is the last argument v of $\mathcal{F}[\![\bullet]\!]$.
Analogously to `WHILE`, we thus define:

$$\begin{aligned}
\mathcal{F}[\![\mathtt{X}]\!]\ B\ v &= v \\
\mathcal{F}[\![\mathtt{nil}]\!]\ B\ v &= \text{nil} \\
\mathcal{F}[\![\mathtt{cons}\ E\ F]\!]\ B\ v &= \langle\, \mathcal{F}[\![\mathtt{E}]\!]\ B\ v . \mathcal{F}[\![\mathtt{F}]\!]\ B\ v \,\rangle \\
\mathcal{F}[\![\mathtt{hd}\ E]\!]\ B\ v &= \begin{cases} d & \text{if } \mathcal{F}[\![\mathtt{E}]\!]\ B\ v = \langle\, d.e \,\rangle \\ \text{nil} & \text{otherwise} \end{cases} \\
\mathcal{F}[\![\mathtt{tl}\ E]\!]\ B\ v &= \begin{cases} e & \text{if } \mathcal{F}[\![\mathtt{E}]\!]\ B\ v = \langle\, d.e \,\rangle \\ \text{nil} & \text{otherwise} \end{cases}
\end{aligned}$$

Function application is defined as:

$$\mathcal{F}[\![\mathtt{f(}E\mathtt{)}]\!]\ B\ v = \begin{cases} \mathcal{F}[\![B]\!]\ B\ d & \text{if } \mathcal{F}[\![E]\!]\ B\ v = d \in \mathbb{D} \\ \perp & \text{otherwise} \end{cases}$$

The semantics of `if` E `then` F `else` G is as usual: first evaluate E. If E evaluates to `nil` evaluate G else evaluate F.

The questions start on the next page.

Questions

1. The (semantics of) language `F` uses the binary tree data type $\mathbb{D}$ we already know from `WHILE`. 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) $\langle\langle\langle \mathrm{nil}.\mathrm{nil}\rangle.\langle \mathrm{nil}.\mathrm{nil}\rangle\rangle.\langle\langle \mathrm{nil}.\mathrm{nil}\rangle.\mathrm{nil}\rangle\rangle$

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

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

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

```
in X out cons nil f(X)
       where  f(X) = if X
                     then cons nil f(tl X)
                     else nil
```

   Your description should explain the result for *any* input $d \in \mathbb{D}$ and should refer to d. Your explanation must not narrate what or how the program executes but describe the function it implements. [10 marks]

3. Let us encode `F`-programs as data by the following encoding that uses lists (which we already know how to encode in $\mathbb{D}$):

$$
\begin{array}{lcl}
\ulcorner \texttt{input X output } E \texttt{ where f(X)=} B \urcorner & = & [\ulcorner E \urcorner, \ulcorner B \urcorner] \\
\ulcorner \texttt{X} \urcorner & = & [\texttt{var}] \\
\ulcorner \texttt{nil} \urcorner & = & [\texttt{quote}, \texttt{nil}] \\
\ulcorner \texttt{cons } E\ F \urcorner & = & [\texttt{cons}, \ulcorner E \urcorner, \ulcorner F \urcorner] \\
\ulcorner \texttt{hd } E \urcorner & = & [\texttt{hd}, \ulcorner E \urcorner] \\
\ulcorner \texttt{tl } E \urcorner & = & [\texttt{tl}, \ulcorner E \urcorner] \\
\ulcorner \texttt{if } E \texttt{ then } F \texttt{ else } G \urcorner & = & [\texttt{if}, \ulcorner E \urcorner, \ulcorner F \urcorner, \ulcorner G \urcorner] \\
\ulcorner \texttt{f(} E \texttt{)} \urcorner & = & [\texttt{appf}, \ulcorner E \urcorner]
\end{array}
$$

   For instance, the data representation of the program `p` in Question 2 is as follows:

```
[
 [cons,[quote,nil],[appf,[var]]] ,
 [if,[var],[cons,[quote,nil],[appf,[tl,[var]]]],
           [quote,nil]
 ]
]
```

Give the data representation of the F-program add given in the description of F on page 2. [10 marks]

4. In the lectures we have introduced WHILE-programs as notion of computability (in other words, WHILE-programs were chosen as "effective procedures"). Would F-programs be a good alternative choice for "effective procedures"? Give reasons for your answer but no formal proof is required. [12 marks]

5. 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:

$$\begin{array}{lcl} concat[[], \mathtt{M}] & = & \mathtt{M} \\ concat[\mathtt{A} :: \mathtt{R}, \mathtt{M}] & = & \mathtt{A} :: concat[\mathtt{R}, \mathtt{M}] \end{array}$$

So, for instance $concat[[\mathtt{1},\mathtt{2},\mathtt{3}],[\mathtt{4},\mathtt{5}]] = [\mathtt{1},\mathtt{2},\mathtt{3},\mathtt{4},\mathtt{5}]$.

(a) Write a *single* WHILE-program that implements *concat* and submit the source code in a file called concat.while. You may call as macro any WHILE-program published on Canvas. You can use extended WHILE-syntax but your program must run correctly in hwhile. [12 marks]

(b) Write a *single* F-program that implements *concat* and submit it in a file called concat.F. [10 marks]

6. An F-interpreter intF written in WHILE is a program for which the following holds:

$$[\![\mathtt{intF}]\!]^{\mathtt{WHILE}}[\ulcorner p \urcorner, d\,] = [\![p]\!]^{\mathtt{F}} d$$

for all F-programs p and input d$\in \mathbb{D}$.

Implement the F-interpreter from above in hwhile. Since hwhile does not have syntax sugar for special atom appf, you need to use something else instead in your code (also for the auxiliary atoms needed for the interpreter), so use @while to encode appf. Get inspiration by looking at the code for the WHILE self-interpreter. Test your interpreter but don't submit the test data. Marking will also use testing so please make sure your program runs

without syntax errror. If there are certain features that you cannot implement then leave those out but still submit a working program. Add comments where indicated to help the marker understand what you're doing. This may be important if your code is not working correctly.

Submit your interpreter together with all macros as *source* code, i.e. `WHILE` programs. Make sure the main program, i.e. the interpreter, is called `intF` and submit it in a file `intF.while`.

Make sure you include programs called as macros that are *not* already published on our Canvas site. If your program is incomplete or does not run for syntactic reasons, it will automatically receive 0 marks! [26 marks]