University of Sussex
Informatics

Spring 2020

Limits of Computation
Feedback for Exercises 6
Dr Bernhard Reus

Self-referencing Programs, Church-Turing Thesis & Time Measure
(Lectures 10–12)

1. Assume we added to `WHILE` an expression `*myself` such that $\mathcal{E}[\![\texttt{*myself}]\!]\,\sigma = \ulcorner p \urcorner$ where p is the program in which this expression is used. This would buy us immediate access to the encoding of a program within this program itself: a powerful programming principle, also called *reflection*.

   (a) In this extended language, write a program that outputs its own encoding (AST). This is now suddenly very easy.

   Answer:

   ```
   quine read X {
    Y := *myself
   }
    write Y
   ```

   (b) Explain why this feature (`*myself`) can be again deemed a "simple" `WHILE`-extension that can be translated away into "core" `WHILE`. In other words, explain why we could rewrite any program using (`*myself`) into a `WHILE`-program (with the same semantics) that does not use this extension.
   Note that you do not have to actually carry out the compilation.

   Answer:
   This is a consequence of the Recursion Theorem. Using

   ```
   p read L {
    q:= hd L;
    d:= hd tl L
   }
   write q
   ```

we get by definition that

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

where the encoding brackets have been explicitly written (often we drop them for simplicity).
Now by the Recursion Theorem we know that because of the above there must be a WHILE-program q such that

$$[\![q]\!]^{\mathtt{WHILE}}(d) = [\![\mathtt{p}]\!]^{\mathtt{WHILE}} \ulcorner [q, d] \urcorner$$

and putting it all together we obtain

$$[\![q]\!]^{\mathtt{WHILE}}(d) = \ulcorner q \urcorner$$

or if dropping the encoding brackets $[\![q]\!]^{\mathtt{WHILE}}(d) = q$.

2. What does the Church-Turing thesis say and what evidence do we have for it?

   Answer:
   The thesis says that all "reasonable" computation models are equivalent (i.e. are of equivalent power). The evidence we have is the fact that for all those models we looked at (and others we know as well, even if we have not discussed them) we can prove that this is the case. We can compile a program (effective procedure) in any (reasonable) model of computation into a program (effective procedure) of the exact same behaviour (same semantics!) in any other one.

3. Consider the following program:

```
prog read X {
 Y:= cons X X
}
write Y
```

   Compute $\mathrm{time}_{\mathtt{p}}^{\mathtt{WHILE}}(d)$ for any input $d \in \mathbb{D}$.

   Answer:
   $\mathrm{time}_{\mathtt{p}}^{\mathtt{WHILE}}(d) = 2 + t$ where $\mathtt{Y} := \mathtt{cons\ X\ X} \vdash^{time} \{\mathtt{X} : d\} \Rightarrow t$.
   Let us compute t, for which need the rule:

$$\mathtt{X} := \mathtt{E} \vdash^{time} \sigma \Rightarrow t + 1 \text{ if } \mathcal{T}\mathtt{E} = t$$

   Instantiating Y for X, $\{\mathtt{X} : d\}$ for σ, cons X X for E we obtain:

$$\texttt{Y} := \texttt{cons X X} \vdash^{time} \{\texttt{X} : d\} \Rightarrow \mathcal{T}\texttt{cons X X} \ + \ 1$$

By the rules for $\mathcal{T}$ that tell us how much time it takes to evaluate WHILE-expressions we get:

$$\mathcal{T}\texttt{cons X X} = 1 + \mathcal{T}\texttt{X} + \mathcal{T}\texttt{X} = 1 + 1 + 1 = 3$$

Therefore, $\text{time}_{\texttt{p}}^{\texttt{WHILE}}(d) = 2 + (3 + 1) = 6$.

4. Write a WHILE-program q with the following time measure:

$$\text{time}_{\texttt{q}}^{\texttt{WHILE}}(d) = 5 \times \text{length}(l_d) + 4$$

where l_d is the list encoded by d, i.e. $d = \ulcorner l \urcorner$ for any $d \in \mathbb{D}$.

Answer: A program that satisfies the requirement is e.g.

```
prog read X {
while X {
 X:= tl X
  }
}
write X
```

Note that the body of the while loop takes 3 units – 1+(1+1) – to execute, but the guard and the check for nil add 2 = 1+1 units. So the loop execution takes 3+2=5 times the length of the input as list. Wait, we have to consider the termination case of the loop, for which 2 more units are needed for the guard evaluation and nil-check, and finally 2 more units for input and output.

5. Let L and M be two timed programming languages with identical data types. Which of the following statements are true? Explain your answer briefly.

   (a) $\texttt{L} \preceq^{\text{lintime}} \texttt{M}$ implies $\texttt{L} \preceq^{\text{lintime-pg-ind}} \texttt{M}$

   Answer: No, the second statement is stronger than the first. See (d) and (e) below for an example where this implication does not hold.

   (b) $\texttt{L} \preceq^{\text{lintime-pg-ind}} \texttt{M}$ implies $\texttt{L} \preceq^{\text{lintime}} \texttt{M}$

   Answer: Yes, the first statement stronger than the second (same constant works for all programs in linear potential slowdown).

(c) $\texttt{WH}^1\texttt{LE} \preceq^{\text{lintime-pg-ind}} \texttt{WHILE}$

Answer: Yes, as every $\texttt{WH}^1\texttt{LE}$-program is by definition already a WHILEprogram, the identity is a compiler and so constant factor 1 works for all programs.

(d) $\texttt{WHILE} \preceq^{\text{lintime}} \texttt{WH}^1\texttt{LE}$

Answer: Yes, the WHILE-program is translated into a WHILE-program with one variable that is a list of the values of all the variables of the original program. Access of the original variable has to be compiled into access of the right position in the list represented by the one variable. The extra time of each access is a constant itself linear in the number of variables. So this slows down the original program by a constant factor.

(e) $\texttt{WHILE} \preceq^{\text{lintime-pg-ind}} \texttt{WH}^1\texttt{LE}$

Answer: No, see previous answer: The extra time of each access is a constant itself linear in the number of variables. The linear factor depends on the number of variables and is therefore *not* program independent.