Solution to Problem 5

Problem. Let us define a function to be $(A \cdot n)$-primitive recursive ($(A \cdot n)$-p.r., for short) if it can be obtained from 0, σ, π_i^k, and a function a function $A(n) \cdot n$ (where $A(n)$ is Ackermann's function) by using composition and primitive recursion. Prove that there exists a computable function which is not $(A \cdot n)$-primitive recursive.

Solution.

First part of the proof. Let us first describe how we can assign, to each $(A \cdot n)$-p.r. function, a natural number that we will call this function's $(A \cdot n)$-*pr-code*. This will be done in several steps.

- By definition, an $(A \cdot n)$-p.r. function is obtained from 0, σ, π_i^k, and $A(n) \cdot n$ by using composition $\circ$ and primitive recursion PR. Thus, each such function can be described by an expression containing these symbols and parentheses (and). For example, addition is described as $PR(0, \sigma \circ \pi_3^3)$.
- Symbols 0 and PR are ASCII symbols, which means that they can be directly typed on a usual computer keyboard. However, symbols σ, π_i^k, and $\circ$ are not. To describe them in ASCII, we can use, e.g., LaTeX, a language specifically designed by renowned computer scientist Donald Knuth to translate mathematical symbols into ASCII. In this language, σ, π_i^k, and $\circ$ are described as follows:

  ```
  \sigma,  \pi^k_i, \circ
  ```

  Similarly, $A(n) \cdot n$ is described as

  ```
  A(n)\cdot n
  ```

- After we use this translation, we get a sequence of ASCII symbols. For example, the expression corresponding to addition takes the form

  ```
  PR(0,\sigma\circ\pi^3_3)
  ```

  According to ASCII, each ASCII symbol is represented in a computer as a sequence of 0s and 1s. For example, P is represented as $50_{16} = 0101\ 0000$, R is represented as $52_{16} = 0101\ 0010$, etc., so the expression for addition takes the form

  $$0101\ 0000\ 0101\ 0010\ \ldots$$

- Finally, we append 1 in front of the resulting sequence of 0s and 1s, and interpret the resulting binary sequence as a binary number. For example, for addition, we will get

  $$1\ 0101\ 0000\ 0101\ 0010\ \ldots$$

  This number is what we will call an $(A \cdot n)$-pr-code of the original $(A \cdot n)$-p.r. function.

Second part of the proof: an important lemma. To prove our result, we will need the following lemma:

Lemma. *There exists an algorithm that, given a natural number c:*

- *checks whether c is an $(A \cdot n)$-pr-code of some $(A \cdot n)$-p.r. function, and*
- *if it is, produced an executable file for computing this function; we will denote this file by f_c.*

How we can prove this lemma. An $(A \cdot n)$-pr-code was obtained from the original expression as follows

expression $\stackrel{\text{LaTeX}}{\longrightarrow}$ ASCII expression $\stackrel{\text{ASCII}}{\longrightarrow}$ binary sequence $\stackrel{\text{append 1}}{\longrightarrow}$ $(A \cdot n)$-pr-code.

Thus, to get back from the $(A \cdot n)$-pr-code to the original $(A \cdot n)$-p.r. function, we need to follow these steps in reverse order:

- First, we strip off the first 1 from the *natural* binary description of the given natural number n, i.e., from a description in which we skip all leading 0s; for example, $n = 5$ is represented as 101, not as 0101. As a result, we get a binary sequence. This step is not possible only in one case: when $n = 0$; in this case, we stop this algorithm and return the answer that $n = 0$ is not an $(A \cdot n)$-pr-code of any $(A \cdot n)$-p.r. function.
- Second, we check whether the resulting binary sequence is indeed a sequence of valid ASCII symbols. This is what computers do all the time.
- Third, we use the LaTeX compiler to check that the corresponding ASCII sequence is a valid LaTeX expression. This is what LaTeX compilers do all the time. If it is a valid LaTeX expression, LaTeX translates in into a sequence of mathematical symbols.
- Finally, we check whether the resulting sequence of mathematical symbols is syntactically correct; e.g.,

  $$PR(0,$$

  is not syntactically correct: we have an opening parenthesis but not a closing one, and there is nothing after the comma. This is what compilers do all the time. If it is syntactically correct, then we can use the same ideas that we used before to translate this expression into the Java code: PR corresponds to the for-loop. etc.

Final step of the proof. Let us now consider the function $f(c)$ which is defined as follows:

- if c is an $(A{\cdot}n)$-pr-code of an $(A{\cdot}n)$-p.r. function, we return $f(c) = f_c(c){+}1$;
- otherwise, if c is *not* an $(A{\cdot}n)$-pr-code of an $(A{\cdot}n)$-p.r. function, we return $f(c) = 0$.

Let is prove that this function is computable but not $(A \cdot n)$-primitive recursive.

Proving that the function $f(c)$ is computable. This proof is straightforward: we just show how this function can be computed. Suppose that we are given a natural number c. Then, to compute $f(c)$, we do the following:

- First, we apply the algorithm $\mathcal{A}$ whose existence is proven by the lemma. This algorithm either tells us that c is not a $(A \cdot n)$-pr-code – in which case we return $f(c) = 0$ – or generates the file f_c.
- If c is an $(A \cdot n)$-pr-code, we apply the executable file f_c to the number c, resulting in the value $f_c(c)$, and then add 1 to the result.

This can be described as follows:

$$\xrightarrow{c} \text{is } c \text{ a } (A \cdot n)\text{-pr-code?} \xrightarrow{\text{yes}} \text{apply } f_c \text{ to } c \xrightarrow{f_c(c)} \text{add 1} \xrightarrow{f_c(c)+1}$$
$$\downarrow \text{no}$$
$$0$$

Proving that the function $f(c)$ is not $(A \cdot n)$-primitive recursive. We will prove this by contradiction. Let us assume that the function $f(c)$ *is* $(A \cdot n)$-primitive recursive. Let c_0 denote its $(A \cdot n)$-pr-code. Then, by definition of f_c as a code that computes the original $(A \cdot n)$-p.r. function, for every possible input n, we have

$$f_{c_0}(n) = f(n).$$

In particular, for $n = c_0$, we have

$$f_{c_0}(c_0) = f(c_0).$$

On the other hand, by definition of the function $f(c)$, since c_0 is a $(A \cdot n)$-pr-code, we get

$$f(c_0) = f_{c_0}(c_0) + 1.$$

By comparing these two equalities, we conclude that $f_{c_0}(c_0) = f_{c_0}(c_0) + 1$, i.e., if we subtract $f_{c_0}(c_0)$ from both sides, that $0 = 1$. This is clearly a contradiction.

The only assumption that we made to get this contradiction is that the function $f(c)$ is $(A \cdot n)$-primitive recursive. Thus, this assumption is wrong, and the above defined function $f(c)$ is not $(A \cdot n)$-primitive recursive.

Conclusion. We have come up with a function $f(c)$ which is computable but not $(A \cdot n)$-primitive recursive. Thus, the result is proven.