Towards Making Theory of Computation Course More Understandable and Relevant: Recursive Functions, For-Loops, and While-Loops

Olga Kosheleva\textsuperscript{1} and Vladik Kreinovich\textsuperscript{2}

Departments of \textsuperscript{1}Teacher Education and \textsuperscript{2}Computer Science University of Texas at El Paso, El Paso, TX 79968, USA
olgak@utep.edu, vladik@utep.edu
1. Theory of Computation Is Useful

- *Results* of theory of computation are useful.
- *Example*: finite automata are a useful technique in designing computer hardware and in designing compilers.
- *Textbooks* describe this usefulness reasonably well.
- However, *proofs* are also useful:
  - to avoid rewriting code, programmers try to make it as *general* as possible;
  - however, *too general* problems are often algorithmically *unsolvable*;
  - in this case, attempts to write a general problem are a *waste of time*;
  - a programmer *must be able to tell* when a general problem is not solvable.
2. Pedagogical Problem

• **Reminder:** students need to be able to tell when a problem is not algorithmically solvable.

• **Important:** they need to understand the existing non-solvability proofs.

• **Goal:** the students will be able to adjust these proofs to the new situations.

• **Problem:** many proofs use abstract notions whose relevance to computing is unclear.

• **It is thus necessary:** to explain the relation between the abstract notions and computer practice.

• **What we do:** we show this relation on the example of recursive functions – one of the first abstract notions.
3. Notion of a Recursive Function: Brief History

- **Origin:** this notion was invented in the 1930s by Alonzo Church.

- **Church’s motivation:** to describe what is computable and what is not computable.

- **Once computers appeared:** many parts of Church’s description were used when designing programming languages.

- **In time,** programming languages and Church’s notations evolved in different directions:
  - in theoretical analysis, we want to have models which are *simple* – to make analysis easier;
  - in programming languages, we want to *add features* if this makes programming simpler.

- **However:** we will show that the theoretical notions can still be related to programming practice.
4. How to Describe a For-Loop in Precise Terms?

• Let us start with a simple program for computing $a^m$:

```java
power = 1;
for(int i = 1; i <= m; i ++) {
    power = power * a;
}
```

• This is not a precise mathematical notation, because:
  - in math, the variable is assumed to have the same value in different parts of the equation, but
  - $power = power * a$ means that $power$ is assigned a new value: the product of the old value and $a$.

• To make the description mathematically precise, we must thus explicitly indicate the iteration number.
5. Describing a For-Loop (cont-d)

- To make the description mathematically precise, we must explicitly indicate the iteration number:
  \[ \text{power}(0) = 1 \]
  \[ \text{power}(i + 1) = \text{power}(i) \times a \]
- The value of the variable `power` also depends on `a`:
  \[ \text{power}(a, 0) = 1 \]
  \[ \text{power}(a, i + 1) = \text{power}(a, i) \times a \]
- In a general for-loop with parameters \( \bar{a} = a_1, \ldots, a_k \) in which a variable \( h \) changes:
  - we first assign some initial value to the variable:
    \[ h(\bar{a}, 0) = f(\bar{a}) \]
  - on each iteration, we use the previous value of \( h, \bar{a}, \) \( i \) to produce a new value:
    \[ h(\bar{a}, i + 1) = g(\bar{a}, i, h(\bar{a}, i)) \]
6. Resulting Description of a For-Loop

- Let \( \bar{a} = a_1, \ldots, a_k \) be a list of parameters.

- To describe a for-loop in which a variable \( h \) changes we need to know:
  - an algorithm \( f(\bar{a}) \) assigning an initial value to \( h \); this value may depend on the parameters \( \bar{a} \);
  - an algorithm \( g(\bar{a}, i, h(\bar{a}, i)) \) that describes what is happening inside the loop, on each iteration.

- Once we have these two algorithms \( f \) and \( g \), we can describe a function \( h \) computed by the for-loop:
  \[
  h(\bar{a}, 0) = f(\bar{a}); \quad h(\bar{a}, i + 1) = g(\bar{a}, i, h(\bar{a}, i)).
  \]

- This is exactly Church’s primitive recursion!

- So, primitive recursion can be described as a natural formalization of a for-loop.
7. Beyond For-Loops: A While-Loop

- In the traditional for-loop, we know beforehand how many iterations we make.

- In some algorithms, we run iterations $x_k$ until the process converges; e.g., to compute $x = \sqrt{a}$:

  $x_0 = 1, \quad x_{k+1} = \frac{1}{2} \cdot \left( x_k + \frac{a}{x_k} \right)$, until $|x_{k+1} - x_k| \leq \varepsilon$.

- In this case, we run a while-loop, which runs until a stopping condition $P$ is satisfied.

- So, to describe while-loops, we need to describe the smallest $m$ for which $P(n, m)$ holds.

- This value $f(n) \overset{\text{def}}{=} \mu m. P(n, m)$ is exactly Church’s μ-recursion!

- Thus, the basic ideas behind recursive functions are exactly natural formalizations of for- and while-loops.
8. Resulting Meaning

- **We have shown:** that
  - primitive recursion corresponds to for-loops, and
  - mu-recursion corresponds to while-loops.

- **From this viewpoint:** many theoretical results acquire natural practical meaning.

- **Example:** the result that not every computable function is primitive recursive.

- This is the first, simplest example of the diagonal construction that is later used in many other proofs.

- **Meaning of the result:**
  - while-loops are needed,
  - because not all computable functions can be computed by using only for-loops.
9. Acknowledgment

This work was supported in part

- by the National Science Foundation grants HRD-0734825 (Cyber-ShARE Center) and DUE-0926721, and

- by Grant 1 T36 GM078000-01 from the National Institutes of Health.