How Can We Use Church-Turing Thesis to Prove Results?

Problem. At first glance, we have what we wanted: we have a formal definition of what is computable. Namely, what is computable is what can be described by a Java program. So, to prove that something is not computable, it is sufficient to prove that no Java program can compute this.

But here is a problem. Java has many features, so many that even in a four-semester sequence, we do not always teach all of them. It has many data types, it have many constructions and operations. To prove that something cannot be computed by a Java program, we really need to show that none of these numerous features will help. With so many features, this is a very difficult task. How can we make it simpler?

How we can make proofs simpler. Good news is that many features of Java have been added to make programming simpler or to make operations more efficient, but they do not add to the ability to compute – and thus, can be safely ignored if we are only interested what is computable and what is not. Let us give a simple example: Java has addition and Java has the `++` operation. From the viewpoint of computability, the operation `a++` is not needed: we can always write

```
a = a + 1;
```

Yes, the using the general addition operation to add 1 is less efficient than the use of the special hardware supported operation "plus 1", but adding the "plus 1" operation to the list of possible operations does not increase the scope of what is computable.

Similarly, Java has addition, subtraction, and unary minus. From the viewpoint of computability, we do not need subtraction if we have two other operations, since $a - b$ can always be represented as $a + (-b)$.

Let us see how, with this in mind, we can simplify Java – and still retain the possibility to compute everything that can be computed in Java.

Let us keep only the simplest data type. Let us start this simplification with data types. Java has many data types. Java is a typed language – it is very picky about data types: to each value, you can only apply operations allowed for the corresponding data type. For example, if you declare a variable of type Boolean, you cannot apply arithmetic operations to this variable – and,

vice versa, if you declare an integer variable x, you cannot directly use in a conditional statement.

In the computer, however, everything is represented as a sequence of 0s and 1s. For example, in the usual compilers, "false" is represented as 0 and "true" as 1. In many other programming languages such as C and C-related ones (such as C++), we can use this fact and we can, e.g., declare an integer and then use it in a conditional statement: 0 will mean "false", positive values will mean "true".

Types are introduced in Java to make programming easier and to help programmers avoid adding-apples-and-oranges-type mistakes: if we accidentally add an integer and a Boolean variable, the compiler will give us an error message. However, types themselves do not add any additional computational ability. Thus, from the viewpoint of what can be computed and what cannot, it is sufficient to simply consider each value as a sequence of 0s and 1s.

The problem with these sequences is that there are not too many Java operations on such sequences. But the situation becomes different – and many operations become possible – if we take into account that each such sequence can be naturally understood as a binary representation of a natural number – i.e., of a non-negative integer. This is exactly how such integers are represented in a computer. So, if we are interested in what is computable and what is not, then it make sense to assume that we only deal with natural numbers

$$0, \quad 1, \quad 2, \quad \ldots$$

Operations on natural numbers. For integers in Java, we have addition, subtraction, multiplication, division, remainder operation, and operations of adding and subtracting 1. Addition, multiplication, division, and remainder can be directly used, since the sum, product, etc., of natural numbers is also a natural number.

With subtraction, we have a minor problem: if we subtract one natural number from another, we may get a negative number: e.g., $2 - 5 = -3 < 0$. What can we do? We can do what kids do in first grade, when they only know non-negative numbers. Namely, in general, subtraction like $5 - 2$ means that if I had 5 dollars on me, and I owed 2 dollars to my friend, then after paying him back I am left with $5 - 2 = 3$ dollars. What if I had 2 dollars on me, and I owe my friend 5 dollars? Then, I give him both dollars that I had – I cannot pay more, since 2 dollars is all I have. So, what I am left with is 0 dollars in my pocket. In this sense, it makes sense to interpret $2 - 5$ as 0.

To distinguish the resulting modified subtraction operation from the actual subtraction, this new operation (limited to natural numbers) is usually described by a dot on top: $a \dot{-} b$. Formally:

- if $a \geq b$, then $a \dot{-} b = a - b$; and
- if $a < b$, then $a \dot{-} b = 0$.

Likewise, instead of the $--$ operation, we can have a similarly modified operation $a \dot{-} \dot{-}$:

- for $a \geq 1$, we have $a \dot{-} \dot{-} = a - 1$, and
- for $a = 0$, we have $0 \dot{-} \dot{-} = 0$.

Notation for ++ in theoretical computer science. In programming, the operation of adding 1 is denoted by ++. However, this is a reasonably recent notation; the corresponding theoretical computer science analysis of what is computable appeared way before this notation was introduced. In this analysis, a different notation was used.

This notation was motivated by the fact that this operation is, from the mathematical viewpoint, a particular case of addition. In mathematics, the sum of several numbers is denoted by the symbol $\sum$, which is the capital Greek letter "sigma" ("s"), the first letter of the word "sum". It is natural to denote a particular case of this operation by a *small* Greek letter "sigma", which looks like this: σ. This is the notation used in theory of computation: e.g., $\sigma(2)$ means $2 + 1$, i.e., 3.

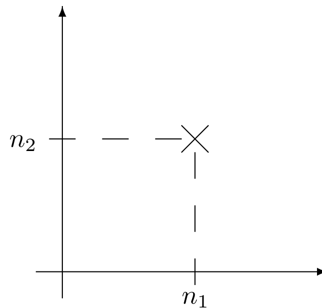
Tuples of natural numbers. In many computations, we need to produce not a single value, but several values: e.g., when we predict weather, we need to predict temperature, humidity, wind speed, etc.

In our terms, this means that we need to consider not a single natural number, but a tuple $\overline{n} = (n_1, \ldots, n_k)$ of natural numbers.

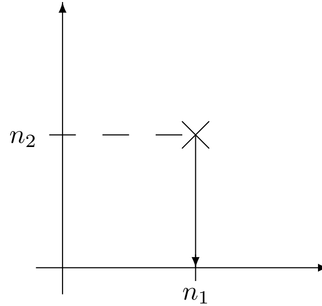
Projection operation. Once we have a tuple $\overline{n} = (n_1, \ldots, n_k)$ of k numbers, then for each i from 1 to k, we need to be able to select the i-th number. This selection operation is known as *projection* and if denoted by π_i^k, where π is the Greek version of the letter p – the first letter of the word "projection":

$$\pi_i^k(n_1, \ldots, n_i, \ldots, n_k) = n_i.$$

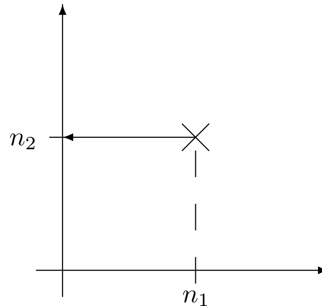
Why is this called projection? Let us look how this operation looks like for $k = 2$. In this case, each tuple $\overline{n} = (n_1, n_2)$ can be represented by a point of the plane, with coordinates n_1 and n_2:



In these terms, computing n_1 is indeed a projection on the n_1-axis:



while computing n_2 is a projection on the n_2-axis:



Composition. If we:

- first apply a function f to the input n getting $a = f(n)$, and
- then apply the function g to the result $f(n)$, getting $g(a) = g(f(n))$,

in mathematics this is called a *composition* and denoted by $g \circ f$.

Composition is naturally computed if we have two Java instructions one after another:

```
int a = f(n);
int y = g(a);
```

Thus, if we can compute two functions, we can also compute their composition as well.

What operations and constructions do we have. We have for-loops, we have while-loops, we have if-then statements. Let us start with for-loops, since without the loops, computations are not possible. Then, we will have the following informal "definition".

Informal "definition". We say that a function is *simple* if it can be obtained from 0, 1, 2, $\ldots$, $+$, σ, $\dot{-}$, $\dot{-}\dot{-}$, $\cdot$, $/$, $\%$, and π_i^k by using composition and the for-loop.

Let us simplify further. From the viewpoint of computability, do we need to also explicitly list 1, 2, ..., i.e., all natural numbers? Not really, since as long as we have 0 and σ:

- we can describe 1 as $\sigma(0)$,
- we can describe 2 as $\sigma(\sigma(0))$ (i.e., in mathematical notations, as $(\sigma \circ \sigma)(0)$), etc.

Thus, we can only keep 0.

Do we need to explicitly list addition? Not really: if we have σ – i.e., the ++ operation – then we can compute $a + b$ by using a for-loop:

```
int c = a;
for(int i = 1; i <= b; i++)
  {c++;}
```

In this program, we start with the value a and add 1 b times. The resulting value

$$a + 1 + 1 + \ldots + 1 \text{ (}b\text{ times)}$$

is clearly equal to $a + b$.

Similarly, it turns out – as we will show later – that all the other operations can be described as long as we have 0, σ, projections, and the for-loop. To describe this idea in precise terms, we need to describe, in precise terms, what a for-loop does. The corresponding formalization is known as *primitive recursion*, and the corresponding functions – obtained by using the for-loop – are known as *primitive recursive.*