Total Functions and Partial Functions

Apr 1, 2022

Given a function and an input for which that function is defined, does the function return an answer? Functions that always return answers are called ***total***, and functions that don't are called ***partial***.

Note that we are only considering **inputs for which the function is defined**. It's easy to define a function on a certain set of inputs (the *domain*) but **forget to cover some cases**. This sort of thing happens all the time in both programming (for instance, writing a function to handle lists but forgetting to deal with the empty list) and math (as when a student writing an inductive proof forgets to cover the base case). It's easy enough to handle. Just **add the missing clauses** and move on. Alternatively, **redefine the domain** of the function so that it excludes the missing case. Either way, there will be no **"holes"** in the modified function.

How could a function fail to return an answer for an input for which it is defined? This problem is essentially retlated to the **termination of unbounded while-loops**. Functions can be broken down into four classes based on their loop bounds:

1. ***Primitive computable***: no unbounded loops required; all loops bounds can be calculated in advance. (Also known as *primitive recurisve*.)
2. ***General computable***: unbounded loops required; loop bounds cannot be calculated in advance, but loops can nevertheless be guaranteed to terminate. (Also known as *general recurisve*.)
3. ***Uncomputable***: unbounded loops required; loops can be guaranteeed not to terminate for some inputs, but which inputs exactly cannot be determined.
4. **Unknown**: unbounded loops required as far as anyone knows, but this has not been proved.

Let's take a look at some examples.

Primitive Computable

Primitive computable functions encompass **pretty much everything that is encountered in day-to-day programming and math**. A good rule of thumb is that if it can be calculated in **Excel**, it's primitive computable. This includes everything from simple arithmetic to solving Go.

Functions in this class can be written using only loops with explicit fixed bounds. Yet it's common to see **real-world code written with unbounded while-loops** even in cases where there is no good reason for doing so.

General Computable

It's not easy come up with functions that are computable but not primitive. Personally I only know of two examples, and they are both named **after their discoverers**.

The first of these is the **Ackermann** function, also known as the **Sudan-Ackermann-Peters** function. It was devised in the 1920s for the specific purpose of showing that there are computable functions that are not primitive computable. Here's a definition in **Python**:

```
def ackermann(bound: int, total: int) -> int:
    stack = []

    while True:
        if bound == 0:
            total += 1

            if not stack:
                return total
```

```
            bound = stack.pop()
            continue

        if total == 0:
            total = 1
            bound -= 1
        else:
            total -= 1
            stack.append(bound - 1)
```

This definition contains an **open while-loop**, and it **cannot be rewritten with a bounded loop**. There is no method for estimating the number of loop passes required short of actually calculating it. Still, it can be shown that it will **always terminate**. On every iteration either `bound` or `total` is decremented. `bound` is increased from time to time, but never past its initial value. The output grows very, very fast with respect to its inputs.

Even faster-growing is the **Goodstein function**, which has to do with representing a number in terms of sums of powers of some base and then incrementing that base. Like the Ackermann function, the Goodstein function can be proved, despite its staggering growth, to be total. But whereas the totality of Ackermann can be proved without too much difficulty, proving the totality of Goodstein is **provably difficult**. Specifically, it requires **infinitary reasoning** and therefore **cannot be proved in Peano arithmetic**.

Uncomputable

Uncomputable functions do not return answers for some inputs. The classic example of uncomputability is the **halting problem**: does a given program halt or not? Any attempt to implement a function to calculate haltingness for arbitrary programs is **doomed** to be either partial or incorrect.

In practice, a reasonable means of coping with uncomputability is to **impose a bound**. Rather than asking if a program halts *ever*, we can ask if it halts within *some number of steps*. This question in contrast