Prof Pitts' 1B CS Computation theory notes: A Discussion of Exercise 10 part (c)

Thomas Forster

April 23, 2020

Two of my supervisees were asking me about the last part of Q 10 on [1]. I think it may be worth setting down in writing the off-the-cuff answer i gave them, and amplifying it.

The question is: show that all the slices of the Ackermann function are primitive recursive. You prove by induction on n that the nth slice is primitive recursive. That is to say, you show how to obtain a primitive recursive declaration for the $n+1$th slice from a primitive recursive declaration for the nth slice. I might write out a proof of this later if there is a call for it. It's fiddly, but there is no great mystery to it.

The next part is: why does this show that the Ackermann function is total recursive? The answer to this is quite subtle. How do I compute, say, ack$(15, 17)$? Well, I reach for the code for the 15th slice and I run it on input 17. Since the 15th slice is primitive recursive it is total (remember that we proved by induction on the recursive datatype of primitive recursive functions that they are all total) so when we run the code for the 15th slice on the input 17 we can be confident of getting an answer.

This prompts the question (which was what I took my supervisee to be asking) Why does this not prove that the Ackermann function is primitive recursive?—and i shall answer that later. However it later occurred to me that what she was not happy about was the expression 'total recursive'. Here 'recursive'—as so often—just means 'computable'. It is also recursive in a more natural sense in that it has a declaration which uses recursion, but is not *primitive* recursive in that it recurses on two variable places simultaneously.

What Prof. Pitts wanted you to say was that, on the basis that all the slices are primitive recursive, one can conclude that one can reliably compute Ackermann on all inputs.

What i took her question to be was: "Why does this not prove that the Ackermann function is *primitive* recursive?" Good question.

The first point to make is that the recursive datatype of primitive recursive functions is closed under `if ... then ... else ...` in the sense that if f and g are primitive recursive functions and p is a predicate whose characteristic function is primitive recursive then

$$\texttt{if } p \texttt{ then } f \texttt{ else } g$$

is primitive recursive too. (I don't know if it is proved in the lectures—there isn't time to prove everything! You might like to try it as an exercise ... it's a standard fact)

Indeed we can clearly stitch together any fixed finite number of primitive recursive functions in this way.

Contrast this with what we are doing when we try to compute Ackermann of x and y. We call up the code for the xth slice of Ackermann, and feed y to it. But there are infinitely many slices! And we can't reach them all with finitely many `branch` commands like the one above. We have to do something like:

> `Input` x `and` y;
> `compute the code for the` x`th slice of Ackermann`;
> `run that code on` y.

. . . which calls the function that accepts input i and outputs code for the ith slice of the Ackermann function. This is a perfectly respectable computable function; it just doesn't happen to be primitive recursive. [proving that it isn't primitive recursive relies on showing that Ackermann itself is not primitive recursive and we don't need to get into why that is so unless you want to.]

This gives us another illustration of the central ambiguity in Computation Theory: a number can be both a data object (input to a computable function) and—simultaneously—a numeric code for a program.

What is being pointed up here is the difference between two ways in which code for functions g, h, . . . might be embedded/encoded/represented inside the code for a function f:

> • In the first case (with the `if` . . . `then` . . . `else` . . .) the code for the embedded functions is hard-coded: code for each of the embedded functions is there *as itself* as it were. This route is available if the number of functions to be embedded is finite.
>
> • In the second case (the case of the Ackermann function we are considering) no code is there as–itself; what we have is code that will output code for each of the embedded function on demand.

Frege, writing about something closely related to this, has a wonderful image that may be of help:

> "*Sie sind in der Tat in den Definitionen enthalten, aber wie die Pflanze im Samen, nicht wie der Balken im Hause.*"

". . . they are indeed contained in the definitions, but rather in the way that plants are contained in seeds, not in the way that timbers are contained in a building."

References

[1] Prof A.M. Pitts: Lecture notes for CS1B Computation Theory at `https://www.cl.cam.ac.uk/teaching/1920/CompTheory/lectures/lecture-10.pdf`.