Lecture 21: General Recursive Functions

μ Operator

MATH230

Te Kura Pāngarau | School of Mathematics and Statistics
Te Whare Wānanga o Waitaha | University of Canterbury

Outline

Bounded Search

Continuing on from the definitions of bounded existential and bounded universal quantification, we introduce a new symbol μ for constructing functions of the form

$$\mu t < yP(\mathrm{x}, \mathrm{t}) = \text{ Smallest } t < y \text{ for which } P(\mathrm{x}, t) = 1$$

So the μ operator takes in a predicate and a bound, and **defines a function** which returns the smallest t which satisfies the predicate. If there is no such t, then the function should return the bound y.

Example Smallest prime less than y and greater than x.
$f(x, y) = \mu t < y[(t \text{ prime}) \wedge (x < t)]$

$$f(6, 4) = 4$$

$$f(4, 6) = 5$$

$\mu t < y$ is Primitive Recursive

Bounded minimisation of a primitive recursive predicate is primitive recursive. Idea: count failures up from 0.

Example

$P(x)$ is a predicate such that $P(0) = 0$, $P(1) = 0$, and $P(2) = 1$.

$\mu t < y$ is Primitive Recursive

Bounded minimisation of a primitive recursive predicate is primitive recursive.

$$\mu t < yP(t) = \sum_{u<y} \prod_{t<u} \chi_{\neg P}(t)$$

Question

Can every function that is “intuitively” computable be defined using the constructions of primitive recursive functions? Is this a good definition of “effective procedure”?

Every step of a primitive recursive function is specified. We do not need any further instruction to carry out the computation of such a function. Furthermore, all primitive recursive functions have a finite number of steps.

Count All Functions

Are all functions $\mathbb{N}^k \to \mathbb{N}$ primitive recursive?

Gödel Codes, Again!

Using a similar technique to Gödel allows us to number the primitive recursive functions. We use powers of primes to associate numbers to the basic primitive recursive functions

- $\#(Z) = 11$
- $\#(S) = 13$
- $\#(\pi_i^k) = (p_{k+6})^{i+1}$

If $\#(g) = a$ and $\#(h) = b$, then $\#(g \circ h) = 2^a 3^b$.

If f is defined by recursion on h with base case g, then we assign the code $\#(f) = 5^a 7^b$

Decoding an integer is a primitive recursive process!

from *Computability*, Epstein and Carnielli.

Conclusion

Just by this counting argument we see that there are more functions than there are primitive recursive functions.

Perhaps the extra functions in our count are not computable?

Cantor's Diagonal Returns

The countability of the primitive recursive functions means we have a computable list of primitive recursive functions of one-variable $f_0, f_1, \ldots, f_n, \ldots$

We consider the function $g(n) = f_n(n) + 1$.

Non-Recursive, but Computable

Consider the function h defined as follows

$$
\begin{aligned}
h(0) &= f_0(0) + 1 \\
h(1) &= f_0(1) + f_1(1) + 1 \\
&\vdots \\
h(n) &= f_0(n) + f_1(n) + \cdots + f_n(n) + 1 \\
&\vdots
\end{aligned}
$$

Question: Is h primitive recursive?

Computation and Primitive Recursion

In this way we see that primitive recursion is different from the intuitive idea we have of an effective procedure.

Others gave different ideas about what effective computation should mean.

Stephen Kleene extended the notion of primitive recursion by adding in an unbounded search operator.

Unbounded-Search

In search of a broader class of functions that we can't diagonalise out of we follow Kleene and drop the bound and define the μ operator.

Given a recursive function $f(y, \mathsf{x})$ we define a new function denoted by μf and defined as

$$(\mu f)(\mathsf{x}) = \min\{t \mid f(t, \mathsf{x}) = 0 \text{ and } f(y, \mathsf{x})\downarrow \forall y < t\}$$

Thus returning the minimum zero of a function.

Is μ Computable?

Total v. Partial Functions

This suggests the following definitions:

We say a function $f : \mathbb{N}^k \rightarrow \mathbb{N}$ is **total** if there is a well-defined output for each input.

If there exist $\mathrm{x} \in \mathbb{N}^k$ for which $f(\mathrm{x})$ is not defined, then we say $f(x)$ is a **partial** function.

The broadening of the allowable constructions yields functions which are not total; that is, computable functions which do not have a well-defined output.

Total v. Partial Functions

If φ is a recursive function, we still write $\varphi(x)$ to denote the process of applying φ to x. However, $\varphi(x)$ may not denote any object; there may not be any output.

If we know $\varphi(x)$ is defined, we denote this by $\varphi(x)\downarrow$

If we know $\varphi(x)$ is undefined, we denote this by $\varphi(x)\not\downarrow$

μ-Recursion

We can enumerate the general recursive functions:

$$\varphi_1(x), \varphi_2(x), \ldots \varphi_n(x), \ldots$$

We can write down $\psi(x) = \varphi_x(x) + 1$.

Engineering Machines

In parallel to these theoretical considerations, physicists and electrical engineers were constructing machines and their components to actually carry out more general computational procedures.

This is a different, equally interesting, story that we will not get to talk about in any great detail. Tutorial 7 gave you a small view into the developments in that direction.

However, to make this class of functions more tangible to our modern perspective, we will note how the general recursive functions relate to modern programming constructs.

In the 1960s Albert Meyer (Complexity Theory pioneer) and Dennis Ritchie (C, Unix) wrote a programming language, designed for an (abstract) register machine, which "implements" this notion of computability.

Loop Programs

Loop, or For, programs are constructed using

- **Var** $= x, y, z, ...$
- Assignments $x := 0$, $x := y + 1$, $x := y$
- Sequential composition $p; q$
- Loop x do p end

Where the Loop x do p is interpreted as

- At the start of the loop x is determined.
- The loop-program p is executed that many times.
- Further changes to x does not change the loop.
- No decrement of x required.

Meyer and Ritchie showed such programs are equivalent to the primitive recursive functions.

Example

Addition: $x := x + y$

```
$x := a$; $y := b$;
loop $y$ do
    $x := x + 1$
end
```

Example

Predecessor: $x := y - 1$

```
$x := 0$; $y := a$; $t = 0$;
loop $y$ do
    $x := t$;
    $t := t + 1$;
end
```

Example

Conditional Execution: If $x = 0$, then (do) p.

```
$x := a$; $t = 1$;
loop $x$ do
    $t := 0$;
end;
loop $t$ do
    $p$;
end
```

Example

If $x = 0$, then p, elif $x = 1$ do q, else do r.

Single Variable Recursion

Suppose $f(x)$ is defined recursively with base case $f(0) = 1$ and $f(x+1) = h(x, f(x))$ such that h is primitive recursive and can be calculated by the Loop-program H. Provide a Loop-program, F, that calculates f.

While Programs

Let us define the while-construct as follows

while $x < y$ do p end

Where the condition $x < y$ is tested every loop and p is executed until the condition is false. At which point the program control passes to the next instruction after the while-loop.

Adding this construct to the loop-programs gives programs that compute the general recursive class of functions.

Note: Loop construct is redundant in the presence of while.

Further Reading

Here are some recommended reading to follow up on the lecture content.

- SEP: Recursive Functions.
- Computability, Richard Epstein.
- Computability Theory, Enderton.
- Lectures on the Philosophy of Mathematics, Joel Hamkins.