# Recursive Functions

MATH230

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

# Outline

# Entscheidungsproblem

In 1928 David Hilbert asked if there could be an “effective procedure" that could decide (Yes/No) whether a sentence of first-order logic is a theorem.

Furthermore, he asked if there could be such a procedure that could determine whether a sentence is a theorem of some first-order theory of mathematics.

With the formal language written down, we now turn to the idea of an “effective procedure" or “algorithm". In order to say whether such a thing exists, we must be precise about what we mean.

# Gödel/ASCII/Unicode

We introduced Gödel numbering as a way of counting the wff of a first-order theory with countably many variables and a finite signature. However, Gödel originally introduced the idea in his proof of the incompleteness theorems.

Gödel did more than just show that there is *some* sentence for which it (and it's negation) can't be proved. He gave an effective procedure for generating such a sentence that one could get their hands on.

Gödel numbering was his way to make wff into objects of computation.

Rather than manipulating strings with some rules of computation, Gödel turned the problem into manipulating numbers with some rules of computation.

# Numbers

We will focus on computations with numbers. Most commonly (in mathematics) we use the decimal system (powers of 10) to represent numbers. However, numbers can be represented as sums of powers of different bases.

For example

- (Decimal) $123_{10} = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$
- (Binary) $123_{10} = 1111011_2$

  $1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
- (Unary) $123_{10} = 11111 \cdots 1111_1$

  $1 \times 1^{123} + 1 \times 1^{122} + \cdots + 1 \times 1^1 + 1 \times 1^0$

# Computation?

What should we mean by an “effective procedure"? What properties do we think such a procedure should have?

# Primitive Recursion: Initial Functions

Primitive recursive functions $f : \mathbb{N}^k \to \mathbb{N}$ are built from the following basic functions.

Zero: $Z(n) = 0$

Successor: $Succ(n) = s(n)$

Projections: $\pi_i^k(x_1, \ldots, x_i, \ldots x_k) = x_i$

These are chosen as there is no questioning the computability of these functions. In all cases it's clear what the output should be.

More sophisticated primitive recursive functions are built inductively from these initial functions according to finitely many applications of function composition and recursion.

# Function Composition

If $g : \mathbb{N}^m \to \mathbb{N}$ is primitive recursive and $h_1, \ldots, h_m : \mathbb{N}^k \to \mathbb{N}$ are each primitive recursive, then the function $f : \mathbb{N}^k \to \mathbb{N}$ defined by function composition

$$f(\mathrm{x}) = g(h_1(\mathrm{x}), \ldots, h_m(\mathrm{x}))$$

is a primitive recursive function.

**Example**

# Recursion: Single Variable

If $g$ is a primitive recursive function and $d \in \mathbb{N}$, then the function $f : \mathbb{N} \to \mathbb{N}$ defined by

$$f(0) = d$$
$$f(s(n)) = g(f(n), n)$$

is also primitive recursive.

**Example**

# Multiplication and Exponentiation

Provide recursive definitions of multiplication and exponentiation.

# Recursion: Multiple Variable

If $g, h$ are primitive recursive functions of multiple variables, then the following function

$$f(0, \mathrm{x}) = g(\mathrm{x})$$
$$f(s(n), \mathrm{x}) = h(f(n, \mathrm{x}), n, \mathrm{x})$$

is also primitive recursive.

# Effective Procedure as Recursion

One interpretation of what it means for a procedure to be effective, is if there exists a primitive recursive function that computes the output.

It's up to us to combine the initial components by composition and recursion to see how powerful this family of functions is.

# Predecessor

Show that the predecessor function is primitive recursive.

# Limited Subtraction (Monus)

Show that limited subtraction is primitive recursive.

# Zero Test

The zero test is primitive recursive.

# Signature

The non-zero test, or signature function, is primitive recursive.

# Absolute Difference

The absolute difference is primitive recursive.

# Min, Max

Min (or max) of two variables is primitive recursive.

# Piecewise Functions

If the primitive recursive functions are to be able to perform all possible computations, then we need to be able to write functions that can deal with conditional branching.

$$f(n) = \begin{cases} 3n+1 & n \text{ odd} \\ \frac{n}{2} & n \text{ even} \end{cases}$$

# Addition of Recursive Functions

Given two functions $f_1, f_2$, we can define the point-wise sum of these functions as follows

$$(f_1 + f_2)(n) = f_1(n) + f_2(n)$$

# Addition of Recursive Functions

Given functions $f_1, f_2, f_3$, we can define the point-wise sum of these functions as follows

$$(f_1 + f_2 + f_3)(n) = f_1(n) + f_2(n) + f_3(n)$$

# Addition of Recursive Functions

Given a primitive recursive function $f$, we can define the point-wise sum of the function with itself finitely many times as follows

$$(f + \cdots + f)(n) = f(n) + \cdots + f(n)$$

# Multiplication of Recursive Functions

Given two functions $f_1, f_2$, we can define the point-wise multiplication of these functions as follows

$$(f_1 \cdot f_2)(n) = f_1(n) \cdot f_2(n)$$

# Multiplication of Recursive Functions

Given functions $f_1, f_2, f_3$, we can define the point-wise product of these functions as follows

$$(f_1 \times f_2 \times f_3)(n) = f_1(n) \times f_2(n) \times f_3(n)$$

# Multiplication of Recursive Functions

Given a primitive recursive function $f$, we can define the point-wise product of the function with itself finitely many times as follows

$$(f \times \cdots \times f)(n) = f(n) \times \cdots \times f(n)$$

# Recursive Predicates

In order to write conditional functions, we need to be able to compute whether the conditions are satisfied i.e. we need primitive recursive definitions of the conditions.

We define the characteristic function of an $n$-ary predicate $P(\mathrm{x})$

$$\chi_P(\mathrm{x}) := \begin{cases} 1 & \text{if } P(\mathrm{x}) \\ 0 & \text{if } \neg P(\mathrm{x}) \end{cases}$$

We say an $n$-ary predicate is primitive recursive if $\chi_P$ is primitive recursive.

# Primitive Recursive Predicates

Conditional programs (piece-wise functions) may branch according to the values of the following predicates

- Less than, less than or equal to (order)
- Greater than, greater than or equal to (order)
- Equal (identity)

# Propositional Connectives

Suppose $P(x)$ and $Q(x)$ are two primitive $n$-ary recursive predicates.

Show that the following are all primitive recursive

$\neg P(x)$

$P(x) \vee Q(x)$

$P(x) \wedge Q(x)$

$P(x) \rightarrow Q(x)$

# Conditional Functions

We may combine the addition, multiplication, and primitive recursive predicates to obtain conditional functionality.

# Conditional Functions

Suppose we have $k$ distinct $n$-ary primitive recursive predicates $A_1, A_2, \ldots, A_k$ such that each x satisfies precisely one of them. We can use such a family of predicates to define piece-wise functions

$$g(\mathrm{x}) := \chi_{A_1}(\mathrm{x}) \cdot f_1(\mathrm{x}) + \chi_{A_2}(\mathrm{x}) \cdot f_2(\mathrm{x}) + \cdots + \chi_{A_k}(\mathrm{x}) \cdot f_k(\mathrm{x})$$

# Existential Quantifier

If $Q(x)$ is a primitive recursive predicate, then bounded search for an integer that satisfies it is primitive recursive. That is to say, the following is primitive recursive.

$$\chi_{\exists y<n Q(y)} = \begin{cases} 1 & \text{if there exists } y < n \text{ such that } Q(y). \\ 0 & \text{otherwise.} \end{cases}$$

Bounded existential quantification is primitive recursive.

# Example

$$\mathrm{div}(x, y) = \begin{cases} 1 & \text{if } x \mid y. \\ 0 & \text{otherwise.} \end{cases}$$

**Details left for tutorial.**

# Universal Quantifier

If $Q(x)$ is a primitive recursive predicate, then checking if all integers less than a specified bound satisfy it is a primitive recursive process.

$$\chi_{\forall y<n Q(y)} = \begin{cases} 1 & \text{if all } y < n \text{ satisfy } Q(y). \\ 0 & \text{otherwise.} \end{cases}$$

Bounded universal quantification is primitive recursive.

# Example

$$\text{prime?}(x) = \begin{cases} 1 & \text{if } x \text{ is prime.} \\ 0 & \text{otherwise.} \end{cases}$$

**Details left for tutorial.**

# Primitive Recursion Language

In a sense we are defining a (programming?) language for a certain class of functions. We are allowed to use the following functions and constructions to build new ones:

Constants

Projections

Arithmetic functions

Sums and products of functions

$\sum_{y<n}$ and $\prod_{y<n}$

Predicates

Propositional logic

Bounded quantification

# Bounded Search

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

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

So the $\mu$ 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.

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

# 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!

# 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 $\mu$ operator.

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

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

Thus returning the minimum zero of a function.

# Is $\mu$ Computable?

# Total v. Partial Functions

This suggests the following definitions:

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

If there exist $x \in \mathbb{N}^k$ for which $f(x)$ 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 $\varphi$ is a recursive function, we still write $\varphi(x)$ to denote the process of applying $\varphi$ 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 $\varphi(x)\not\downarrow$

# $\mu$-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 1 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** $= x, y, z, \ldots$

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

$$\text{while } x < y \text{ do } p \text{ end}$$

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.