Introduction to the Theory of Computation

Jingde Cheng

Saitama University

Introduction to the Theory of Computation

- Enumerability and Diagonalization
- Finite Automata and Regular Languages
- Pushdown Automata and Context-Free Languages
- Computation: Turing Machines
- Computation: Turing-Computability (Turing-Decidability)
- Computation: Reducibility and Turing-Reducibility
- Computation: Recursive Functions
- Computation: Recursive Sets and Relations
- Equivalent Definitions of Computability
- Advanced Topics in Computability Theory
- Computational Complexity
- Time Complexity
- Space Complexity
- Intractability
- Advanced Topics in Complexity Theory

The Theory of Recursive Functions: What is It? [SEP]

♣ The recursive functions

- The recursive functions are a class of functions on the natural numbers studied in computability theory, a branch of contemporary mathematical logic which was originally known as recursive function theory.
- Such functions take their name from the process of recursion by which the value of a function is defined by the application of the same function applied to smaller arguments.

♣ A characteristic feature of recursive definitions

- A characteristic feature of recursive definitions is that they allow for the values of functions which they describe to be calculated by successively "unwinding" the clause for x > 0 [f(x)] until the clause for x = 0 [f(x)] (the so-called base case) is reached.

The Theory of Recursive Functions: Historical Background [SEP]

♣ The early history of recursive definitions

- Grassmann (1861) and Peirce (1881) first gave the familiar recursive definitions of addition and multiplication.
- The first person to employ the expression "definition by recursion" appears to have been Dedekind in his essay "Was sind und was sollen die Zahlen" (1888).
- Dedekind 1888 first used the notion to obtain functions needed in his formal analysis of the concept of natural number.

The Theory of Recursive Functions: Historical Background [SEP]

♣ The origins of primitive recursion

- In logic, Skolem 1923 noted that many basic functions can be defined by simple applications of the recursion method.
- The first work devoted exclusively to recursive definability was Skolem's (1923) paper: "The foundations of elementary arithmetic established by the recursive mode of thought, without the use of apparent variables ranging over infinite domains."
- This work is significant with respect to the subsequent development of computability theory for at least three reasons. First, it contains a informal description of what we now call the primitive recursive functions. Second, it can be regarded as the first place where recursive definability is linked to effective computability. And third, it demonstrates that a wide range of functions and relations are primitive recursive in a manner which anticipates Gödel's (1931) use of primitive recursion for the arithmetization of syntax.

The Theory of Recursive Functions: Historical Background [SEP]

♣ The next important steps in the development of a general theory of recursive function

- The next important steps in the development of a general theory of recursive function arose as a consequence of the interaction between Hilbert's Program and Gödel's (1931) proof of the Incompleteness Theorems.
- Hilbert (1923) first described a logical calculus for finitary number theory including "recursion and intuitive induction for finite totalities". Although this presentation also included a discussion of definition by simultaneous recursion, a more extensive treatment of what we would now recognize as recursion schemes is given in his well known paper "On the infinite" (1926).
- Gödel (1931) provided the more precise formulation of primitive recursion.

The Theory of Recursive Functions: Historical Background [SEP]

♣ **Arithmetical representability and Gödel's first incompleteness theorem**

◆ **Gödel (1931, On formally undecidable propositions of Principia mathematica and related systems I) provided the more precise formulation of primitive recursion.**

Gödel's (1931, 157–159) definition was as follows:

A number-theoretic function $\phi(x_1, \ldots, x_n)$ is said to be *recursively defined in terms of* the number-theoretic functions $\psi(x_1, x_2, \ldots, x_{n-1})$ and $\mu(x_1, x_2, \ldots, x_{n+1})$ if

(6) i. $\phi(0, x_2, \ldots, x_n) = \psi(x_2, \ldots, x_n)$
ii. $\phi(k+1, x_2, \ldots, x_n) = \mu(k, \phi(k, x_2, \ldots, x_n), x_2, \ldots, x_n)$

holds for all $x_2, \ldots, x_n, k$.

A number-theoretic function ϕ is said to be *recursive* if there is a finite sequence of number-theoretic functions $\phi_1, \phi_2, \ldots \phi_n$ that ends with ϕ and has the property that every function ϕ_k of the sequence is recursively defined in terms of two of the preceding functions, or results from any of the preceding functions by substitution, or, finally, is a constant or the successor function $x + 1$.... A relation $R(x_1, \ldots, x_n)$ between natural numbers is said to be *recursive* if there is a recursive function $\phi(x_1 \ldots, x_n)$ such that, for all $x_1, x_2, \ldots, x_n$

The Theory of Recursive Functions: Historical Background [SEP]

♣ **Arithmetical representability and Gödel's first incompleteness theorem**

◆ **Putting aside Gödel's use of the term "recursive" rather than "primitive recursive", this exposition comes close to coinciding with the contemporary definition of the primitive recursive functions.**

◆ **"Recursive functions have the important property that, for each given set of values of the arguments, the value of the function can be computed by a finite procedure. Similarly, recursive relations (classes) are decidable in the sense that, for each given n-tuple of natural numbers, it can be determined by a finite procedure whether the relation holds or does not hold (the number belongs to the class or not), since the representing function is computable." [Gödel, 1934].**

Characteristics of Recursive Functions

♣ **Characteristics of recursive functions**

◆ **The recursive functions are characterized by the process in virtue of which the value of a function for some argument is defined in terms of the value of that function for some other (in some appropriate sense "smaller") arguments, as well as the values of certain other functions.**

♣ **The basic/initial functions** 基础/初始函数

◆ **In order to get the whole process started a certain class of functions need to be singled out, whose values do not in turn depend of their values for smaller arguments. These are called the basic/initial functions.**

Representing the Natural Numbers by the Primeval Monadic/Tally Notation

♣ **The primeval monadic/tally notation**

◆ **A positive integer *n* is represented by *n* strokes.**

♣ **Representing the natural numbers**

◆ **The variation is needed because we want to consider not just positive integers (excluding zero) but the natural numbers (including zero).**

◆ **We adopt the system in which the number zero is represented by the cipher 0, and a natural number $n > 0$ is represented by the cipher 0 followed by a sequence of *n* little raised strokes or accents.**

◆ **Thus the numeral for one is 0′, the numeral for two is 0″, the numeral for *n* is 0′···′(*n* times), and so on.**

Primitive Recursive Functions: Basic Functions [L-ToC-17]

♣ **Basic/Initial primitive recursive functions** 基础/基本递归函数

Primitive Recursive Functions

To keep the discussion simple, we will consider only functions of one or two variables, whose domain is either I, the set of all nonnegative integers, or $I \times I$, and whose range is in I. In this setting, we start with the basic functions:

1. The **zero function** $z(x) = 0$, for all $x \in I$.
2. The **successor function** $s(x)$, whose value is the integer next in sequence to x, that is, in the usual notation, $s(x) = x + 1$.
3. The **projector functions**

$$p_k(x_1, x_2) = x_k, \qquad k = 1, 2.$$

Primitive Recursive Functions: Basic/Initial Functions

♣ **The zero function**

◆ **The *zero function*, whose value $z(x)$ is the same, namely zero, for any argument x such that $z(0) = 0$, $z(0') = 0$, $z(0'') = 0$, ····.**

♣ **The successor function**

◆ **The *successor function* $s(x)$ [x'], whose value for any number x is the next larger number such that $s(0) = 0'$, $s(0') = 0''$, $s(0'') = 0'''$, ····.**

♣ **Effectively computable functions** 有效可计算函数

◆ **The zero function and successor function are *effectively computable*. They can be computed in one step, at least on one way of counting steps.**

可在一步，或有可数步数

Primitive Recursive Functions: Basic/Initial Functions

♣ **The identity/projection functions of one argument**

- ◆ **The *identity/projection function* of one argument, id or more fully id_1^1, assigns to each natural number as argument that same number as value: $\mathrm{id}_1^1(x) = x$.**

♣ **The identity/projection functions of two arguments**

- ◆ **There are two identity/projection functions of two arguments: id_1^2 and id_2^2.**
- ◆ **For any pair of natural numbers as arguments, these pick out the first and second, respectively, as values: $\mathrm{id}_1^2(x, y) = x$, $\mathrm{id}_2^2(x, y) = y$.**

指定参数数量
指定投到第几个参数

Primitive Recursive Functions: Basic Functions

♣ **The identity/projection functions of *n* arguments**

- ◆ **In general, for each positive integer *n*, there are *n* identity/projection functions of *n* arguments, which pick out the first, second, . . . , and *n*th of the arguments: $\mathrm{id}_i^n(x_1, ..., x_i, ..., x_n) = x_i$.**
- ◆ **Identity functions are also called *projection functions*.**
- ◆ **In terms of analytic geometry, $\mathrm{id}_2^1(x, y)$ and $\mathrm{id}_2^2(x, y)$ are the projections x and y of the point (x, y) to the *X*-axis and to the *Y*-axis respectively.**

♣ **Effectively computable functions**

- ◆ **The identity/projection functions are *effectively computable*. They can be computed in one step, at least on one way of counting steps.**

Primitive Recursive Functions: Basic Functions [SEP]

♣ **Primitive Recursive functions (PR)** 原始递归函数

- ◆ **The *primitive recursive functions*, denoted as PR, constitute a mathematically well-defined class of functions on the natural numbers.**

♣ **Basic/Initial primitive recursive functions**

- ◆ **The basic/initial primitive recursive functions include the null-ary zero function, unary successor function, and *k*-ary identity/projection functions.**

In the case of the primitive recursive functions **PR**, the initial functions include the nullary *zero function* **0** which returns the value 0 for all inputs (and can thus be treated as a constant symbol), $s(x)$ denotes the unary *successor function* $x \mapsto x + 1$, and π_i^k denotes the k-ary *projection function* on to the ith argument (where $0 \leq i < k$)—i.e.,

$$\pi_i^k(x_0, \ldots, x_i, \ldots x_{k-1}) = x_i$$

This class of functions will be denoted by $I_{\mathbf{PR}} = [\mathbf{0}, s, \pi_i^k]$. Note that since π_i^k is a distinct function for each $i, k \in \mathbb{N}$, $I_{\mathbf{PR}}$ already contains infinitely many functions.

Primitive Recursive Functions: Building Complicated Functions [L-ToC-17]

♣ **Building complicated functions from basic primitive recursive functions**

There are two ways of building more complicated functions from these:

1. **Composition**, by which we construct 合成

$$f(x, y) = h(g_1(x, y), g_2(x, y))$$

from defined functions g_1, g_2, h.

2. **Primitive recursion**, by which a function can be defined recursively through 原始递归

$$f(x, 0) = g_1(x), \quad \text{出发点}$$
$$f(x, y+1) = h(g_2(x, y), f(x, y)), \quad \rightarrow$$

from defined functions g_1, g_2, and h.

Primitive Recursive Functions: Definition [L-ToC-17]

♣ **Primitive recursive functions (PR)**

- ◆ **Functions obtainable from the basic/initial primitive recursive functions (zero, successor, and identity/projection functions) by composition and primitive recursion are called *primitive recursive*.**
- ◆ **All primitive recursive functions are effectively computable.**

DEFINITION 13.1

A function is called **primitive recursive** if and only if it can be constructed from the basic functions z, s, p_k, by successive composition and primitive recursion.

Primitive Recursive Functions: Building Complicated Functions

♣ **The composition/substitution operation**

- ◆ **If *f* is a function of *m* arguments and each of $g_1, \ldots, g_m$ is a function of *n* arguments, then the function obtained by *composition/substitution* from $f, g_1, ..., g_m$ is the function *h* of *n* arguments: $h(x_1, ..., x_n) = f(g_1(x_1, ..., x_n), ..., g_m(x_1, ..., x_n))$ (Cn) In shorthand, $h = \mathrm{Cn}[f, g_1, ..., g_m]$.**
- ◆ **If g_i are all effectively computable and *f* is effectively computable, then so is the function *h*.**
- ◆ **The number of steps needed to compute $h(x_1, ..., x_n)$ will be the sum of the number of steps needed to compute $y_1 = g_1(x_1, ..., x_n)$, the number needed to compute $y_2 = g_2((x_1, ..., x_n)$, and so on, plus at the end the number of steps needed to compute $f(y_1, \ldots, y_m)$.**

Primitive Recursive Functions: Building Complicated Functions [SEP]

♣The composition/substitution operation

The functionals of **PR** are those of *composition* and *primitive recursion*. Composition takes j functions $g_0, \ldots, g_{j-1}$ of arity k and a single function f of arity j and returns their *composition*—i.e., the function

$$h(x_0, \ldots, x_{k-1}) = f(g_0(x_0, \ldots, x_{k-1}), \ldots, g_{j-1}(x_0, \ldots, x_{k-1}))$$

of type $\mathbb{N}^k \to \mathbb{N}$. As an example, suppose that f is the multiplication function $mult(x, y)$, g_0 is the constant 3 function (which we may think of as implicitly taking a single argument), and $g_1(x)$ is the successor function $s(x)$. Then the composition of f with g_0 and g_1 is the unary function $h(x) = f(g_0(x), g_1(x)) = mult(3, s(x))$which we would conventionally denote by $3 \times (x + 1)$.

The operation of composition may be understood as a class of functionals which for each $j, k \in \mathbb{N}$ takes as inputs j functions $g_0, \ldots, g_{j-1}$ of arity k and a single function f of arity j and returns as output the k-ary function h which composes these functions in the manner just illustrated. This is described by the following scheme:

Definition 2.1: Suppose that $f : \mathbb{N}^j \to \mathbb{N}$ and $g_0, \ldots, g_{j-1} : \mathbb{N}^k \to \mathbb{N}$ Then the term $Comp^j_k[f, g_0, \ldots, g_{j-1}]$denotes the function

$$f(g_0(x_0, \ldots, x_{k-1}), \ldots, g_{j-1}(x_0, \ldots, x_{k-1}))$$

of type $\mathbb{N}^k \to \mathbb{N}$.

Primitive Recursive Functions: Examples [BBJ-ToC-07]

♣Constant functions

- $h(x) = f(g(x))$ (Cn); $h = \text{Cn}[f, g]$; Let $f = s$, $g = z$

6.1 Example (Constant functions). For any natural number n, let the *constant* function const_n be defined by $\text{const}_n(x) = n$ for all x. Then for each n, const_n can be obtained from the basic functions by finitely many applications of composition. For, const_0 is just the zero function z, and $\text{Cn}[s, z]$ is the function h with $h(x) = s(z(x)) = s(0) = 0' = 1 = \text{const}_1(x)$ for all x, so $\text{const}_1 = \text{Cn}[s, z]$. (Actually, such notations as $\text{Cn}[s, z]$ are genuine function symbols, belonging to the same grammatical category as h, and we could have simply written $\text{Cn}[s, z](x) = s(z(x))$ here rather than the more longwinded 'if $h = \text{Cn}[s, z]$, then $h(x) = z(x)'$'.) Similarly $\text{const}_2 = \text{Cn}[s, \text{const}_1]$, and generally $\text{const}_{n+1} = \text{Cn}[s, \text{const}_n]$.

例 6.1 （常函数）对于任意自然数 n，令常函数 const_n 定义为 $\text{const}_n(x) = n$（对于所有 x）。对于每个 n，从基本函数出发，有限次地使用复合就能够得到 const_n。因为 const_0 就是零函数 z，设 $\text{Cn}[s, z]$是函数 h，于是对于所有 x，$h(x) = s(z(x)) = s(0) = 0' = 1 = \text{const}_1(x)$，因此，$\text{const}_1 = \text{Cn}[s, z]$。（实际上，像 $\text{Cn}[s, z]$这样的记号是真正的函数符号，与 h 属于同样的语法范畴，能够将它简单地记为 $\text{Cn}[s, z](x) = s(z(x))$，而不用再啰嗦地说“如果 $h = \text{Cn}[s, z]$，则 $h(x) = z(x)''$。）类似地，$\text{const}_2 = \text{Cn}[s, \text{const}_1]$，一般地，$\text{const}_{n+1} = \text{Cn}[s, \text{const}_n]$。

Primitive Recursive Functions: Building Complicated Functions

♣The primitive recursion operation: the simple case

- If f and g are two functions, then the function obtained (be definable) by *primitive recursion* from f and g is the function h where the following equations -- called the *recursion equations* for the function h -- hold:
  $h(x, 0) = f(x)$, $h(x, y') = g(x, y, h(x, y))$ (Pr)
  In shorthand, $h = \text{Pr}[f, g]$.
- If f and g are all effectively computable, then so is the function h.
- The number of steps needed to compute $h(x, y)$ will be the sum of the number of steps needed to compute $z_0 = f(x) = h(x, 0)$, the number needed to compute $z_1 = g(x, 0, z_0) = h(x, 1)$, the number needed to compute $z_2 = g(x, 1, z_1) = h(x, 2)$, and so on up to $z_y = g(x, y - 1, z_{y-1}) = h(x, y)$.

Primitive Recursive Functions: Building Complicated Functions

♣The primitive recursion operation: the general (full) case

- If f is a function of n arguments and g is a function of n+2 arguments, then the function obtained (be definable) by *primitive recursion* from f and g is the function h of n+1 arguments where the following equations -- called the *recursion equations* for the function h -- hold:
  $h(x_1, ..., x_n, 0) = f(x_1, ..., x_n)$,
  $h(x_1, ..., x_n, y') = g(x_1, ..., x_n, y, h(x_1, ..., x_n, y))$ (Pr)
  In shorthand, $h = \text{Pr}[f, g]$.
- If f and g are all effectively computable, then so is the function h.

The Composition and Primitive Recursion Operations

♣The composition/substitution operation

- $h(x_1, ..., x_n) = f(g_1(x_1, ..., x_n), ..., g_m(x_1, ..., x_n))$ (Cn)
- $h = \text{Cn}[f, g_1, ..., g_m]$, $h(x^{\rightarrow}) = \text{Cn}[f, g_1, ..., g_m](x^{\rightarrow})$

♣The primitive recursion operation

- $h(x_1, ..., x_n, 0) = f(x_1, ..., x_n)$,
  $h(x_1, ..., x_n, y') = g(x_1, ..., x_n, y, h(x_1, ..., x_n, y))$ (Pr)
- $h = \text{Pr}[f, g]$, $h(x^{\rightarrow}) = \text{Pr}[f, g](x^{\rightarrow})$

♣Abbreviation

- $x^{\rightarrow} = x_1, \ldots, x_k$

Primitive Recursive Functions [W-ToC-12]

Definition 3.3.1. The class of *primitive recursive functions* is the smallest class $\mathcal{C}$ of functions such that the following hold:

(i) The successor function $S(x) = x + 1$ is in $\mathcal{C}$.

(ii) All constant functions $M^n_m(x_1, x_2, \ldots, x_n) = m$ for $n, m \in \mathbb{N}$ are in $\mathcal{C}$.

(iii) All projection (or identity) functions $P^n_i(x_1, x_2, \ldots, x_n) = x_i$ for $n \geq 1$, $1 \leq i \leq n$, are in $\mathcal{C}$.

(iv) (Composition, or substitution.) If $g_1, g_2, \ldots, g_m, h$ are in $\mathcal{C}$, then

$$f(x_1, \ldots, x_n) = h(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$$

is in $\mathcal{C}$, where the g_i are functions of n variables and h is a function of m variables.

(v) (Primitive recursion, or just recursion.) If $g, h \in \mathcal{C}$ and $n \geq 0$, then the function f defined below is in $\mathcal{C}$:

$$f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$$

$$f(x_1, \ldots, x_n, y + 1) = h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y))$$

where g is a function of n variables and h a function of $n + 2$ variables.

Primitive Recursive Functions: Building Complicated Functions [SEP]

♣ The primitive recursion operation: the simplest case

Primitive recursion is also a functional operation. In the simplest case, it operates by taking a single unary function $g(x)$ and a natural number $n \in \mathbb{N}$ and returns the unary function defined by

(14)
$$h(0) = n$$
$$h(x+1) = g(h(x))$$

In such a definition, the first clause (known as the *base case*) determines the value of h at 0, while the second clause determines how its value at $x+1$ depends on its value at x. In this case it is easy to see that the value of x determines how many times the function g is *iterated* (i.e., applied to itself) in determining the value of h. For instance, if $n = 3$ and $g(x) = mult(x, x)$, then $h(x) = 3^{x+1}$—i.e., the $x+1$st iterate of the map $x \mapsto 3 \times x$.

Primitive Recursive Functions: Building Complicated Functions [SEP]

♣ The primitive recursion operation: the full scheme

The full primitive recursion scheme generalizes (14) in two ways. First, it allows the value of the function h at $x+1$ to depend not just on its own value at x, but also on the value of the variable x itself. This leads to the scheme

(15)
$$h(0) = n$$
$$h(x+1) = g(x, h(x))$$

A second possible generalization to (14) results from allowing the value of h to depend on a finite sequence of auxiliary variables known as *parameters* which may also be arguments to the base case. In the case of a single parameter x, this leads to the scheme

(16)
$$h(x, 0) = f(x)$$
$$h(x, y+1) = g(x, h(x, y))$$

Primitive Recursive Functions: Building Complicated Functions [SEP]

♣ The primitive recursion operation: the full scheme

The addition function $add(x, y)$ may, for instance, be defined in this way by taking $f(x_0) = x_0$ and $g(x_0, x_1) = s(x_1)$. This definition can also be thought of as specifying that the sum $x + y$ is the value obtained by iterating the application of the successor function y times starting from the initial value x in the manner of (14). Similarly, $mult(x, y)$ may be defined by taking $f(x_0) = 0$ and $g(x_0, x_1) = add(x_0, x_1)$. This defines the product $x \times y$ as the value obtained by iterating the function which adds x to its argument y times starting from the initial value 0.

Such definitions may thus be understood to provide algorithms for computing the values of the functions so defined.[20] For observe that each natural number n is either equal to 0 or is of the form $m+1$ for some $m \in \mathbb{N}$. If we now introduce the abbreviation $\overline{n} = s(s(s \ldots (s(\mathbf{0})))) n$-times, the result of applying the successor function s to a number denoted by $\overline{n}$ thus yields the number denoted by $n+1$. We may thus compute the value of $x + y$ using the prior recursive definition of addition as follows:

(17)
$$\begin{aligned} add(\overline{2}, \overline{3}) &= s(add(\overline{2}, \overline{2})) \\ &= s(s(add(\overline{2}, \overline{1}))) \\ &= s(s(s(add(\overline{2}, \overline{0})))) \\ &= s(s(s(\overline{2}))) \\ &= s(s(s(s(s(\mathbf{0}))))) \\ &= \overline{5} \end{aligned}$$

Primitive Recursive Functions: Building Complicated Functions [SEP]

♣ The primitive recursion operation: the full scheme

The full definition of the primitive recursion operation combines both generalizations of (14) into a single scheme which takes as arguments a k-ary function f, a $k+2$-ary function g, and returns a $k+1$-ary function h defined as follows

(18)
$$h(x_0, \ldots, x_{k-1}, 0) = f(x_0, \ldots, x_{k-1})$$
$$h(x_0, \ldots, x_{k-1}, y+1) = g(x_0, \ldots, x_{k-1}, y, h(x_0, \ldots, x_{k-1}, y))$$

Here the first k arguments $x_0, \ldots, x_{k-1}$ to g are the parameters, the $k+1$st argument y is the *recursion variable*, and the $k+2$nd argument $h(x_0, \ldots, x_{k-1}, y)$ gives the prior value of h. An elementary set theoretic argument shows that for each $k \in \mathbb{N}$, if f is k-ary and g is $k+2$-ary, then a there is a unique $k+1$-ary function h satisfying (18)—see, e.g., (Moschovakis 1994, ch. 5).

It will again be useful to introduce a formal scheme for referring to functions defined in this manner:

Definition 2.2: Suppose that $f : \mathbb{N}^k \to \mathbb{N}$ and $g : \mathbb{N}^{k+2} \to \mathbb{N}$. Then the term $\mathcal{P}rimRec_k[f, g]$ denotes the unique function of type $\mathbb{N}^{k+1} \to \mathbb{N}$ satisfying (18).

Primitive Recursive Functions (PR) [SEP]

♣ The primitive recursive functions (PR)

- **We may now formally define the class PR of primitive recursive functions as follows:**

Definition 2.3: The class of *primitive recursive functions* **PR** is the smallest class of functions containing the initial functions $I_{\mathbf{PR}} = \{\mathbf{0}, s, \pi_i^k\}$ and closed under the functionals

$$Op_{\mathbf{PR}} = \{Comp_j^i, \mathcal{P}rimRec_k\}.$$

♣ The primitive recursive functions (PR): Another definition

The foregoing definition specifies **PR** as the *minimal closure* of $I_{\mathbf{PR}}$ under the functions in $Op_{\mathbf{PR}}$. In other words, **PR** may be equivalently defined as the subclass of $\bigcup_{k \in \mathbb{N}} (\mathbb{N}^k \to \mathbb{N})$ satisfying the following properties:

(19)
i. $I_{\mathbf{PR}} \subseteq \mathbf{PR}$
ii. For all $j, k \in \mathbb{N}$ and $f, g_0, \ldots, g_{k-1} \in \mathbf{PR}$ if f is j-ary and g_i is k-ary (for $1 \leq i \leq n$) then $Comp_k^j[f, g_0, \ldots, g_{j-1}] \in \mathbf{PR}$
iii. For all $k \in \mathbb{N}$ and $f, g \in \mathbf{PR}$, if f is k-ary and g is $k+2$-ary then $\mathcal{P}rimRec_k[f, g] \in \mathbf{PR}$
iv. No functions are members of **PR** unless they can be defined by i–iii.

Primitive Recursive Functions (PR) [SEP]

♣ The primitive recursive functions (PR) is countable

Another consequence of the fact that *every* $f \in \mathbf{PR}$ is defined by a term given in this manner by (19) is the following:

Proposition 2.1: The class of functions **PR** is countable.

This can be demonstrated by showing that it is possible to enumerate **PR** as $f_0, f_1, f_2, \ldots$ by introducing a Gödel numbering of terms formed from the expressions $\mathbf{0}, s, \pi_i^k, Comp_k^j$ and $\mathcal{P}rimRec_k$ in the manner described in Section 1.3. Since there are uncountably many functions of type $\mathbb{N}^k \to \mathbb{N}$ for all $k > 0$, this observation also provides a non-constructive demonstration that there exist number theoretic functions which are not primitive recursive.

♣ There exist number theoretic functions which are not primitive recursive

- **Because there are uncountably many functions of type $N^k \to N$ for all $k > 0$, but the class of functions PR is countable, there exist number theoretic functions which are not primitive recursive.**

Primitive Recursive Functions (PR) [SEP]

♣ **The primitive recursive relations**

- **With the definition of PR in place, we may also define what it means for a relation $R \subseteq \mathbb{N}^k$ to be primitive recursive:**

Definition 2.4: $R \subseteq \mathbb{N}^k$ is a *primitive recursive relation* just in case its characteristic function

$$\chi_R(x_0, \ldots, x_{k-1}) = \begin{cases} 1 & \text{if } R(x_0, \ldots, x_{k-1}) \\ 0 & \text{if } \neg R(x_0, \ldots, x_{k-1}) \end{cases}$$

is a primitive recursive function.

Definition 2.4 thus conventionalizes the characterization of a primitive recursive relation $R \subseteq \mathbb{N}^k$ as one for which there exists an algorithm similar to that illustrated above which returns the output 1 on input $\vec{n}$ if R holds of $\vec{n}$ and the output 0 if R does not hold of $\vec{n}$. As will become clear below, most sets and relations on the natural numbers which are considered in everyday mathematics—e.g., the set *PRIMES* of prime numbers or the relation

$$DIV = \{\langle n, m\rangle : n \textit{ divides } m \textit{ without remainder}\}$$

—are primitive recursive.

- **Abbreviation:** $\vec{x}\,(x^{\rightarrow}) = x_1, \ldots, x_k$

Primitive Recursive Functions: Examples [L-ToC-17]

EXAMPLE 13.1

Addition of integers x and y can be implemented with the function $add\,(x, y)$, defined by

$$\begin{aligned} add\,(x, 0) &= x, \\ add\,(x, y+1) &= add\,(x, y) + 1. \end{aligned}$$

To add 2 and 3, we apply these rules successively:

$$\begin{aligned} add\,(3, 2) &= add\,(3, 1) + 1 \\ &= (add\,(3, 0) + 1) + 1 \\ &= (3 + 1) + 1 \\ &= 4 + 1 = 5. \end{aligned}$$

EXAMPLE 13.2

Using the *add* function defined in Example 13.1, we can now define multiplication by

$$\begin{aligned} mult\,(x, 0) &= 0, \\ mult\,(x, y+1) &= add\,(x, mult\,(x, y)). \end{aligned}$$

Formally, the second step is an application of primitive recursion, in which h is identified with the *add* function, and $g_2\,(x, y)$ is the projector function $p_1\,(x, y)$.

Primitive Recursive Functions: Examples [L-ToC-17]

EXAMPLE 13.3

Substraction is not quite so obvious. First, we must define it, taking into account that negative numbers are not permitted in our system. A kind of subtraction is defined from usual subtraction by

$$\begin{aligned} x \dot{-} y &= x - y \text{ if } x \geq y, \\ x \dot{-} y &= 0 \text{ if } x < y. \end{aligned}$$

The operator $\dot{-}$ is sometimes called the **monus**; it defines subtraction so that its range is I.

Now we define the predecessor function

$$\begin{aligned} pred\,(0) &= 0, \\ pred\,(y+1) &= y, \end{aligned}$$

and from it, the subtracting function

$$\begin{aligned} subtr\,(x, 0) &= x, \\ subtr\,(x, y+1) &= pred\,(subtr\,(x, y)). \end{aligned}$$

To prove that $5 - 3 = 2$, we reduce the proposition by applying the definitions a number of times:

$$\begin{aligned} subtr\,(5, 3) &= pred\,(subtr\,(5, 2)) \\ &= pred\,(pred\,(subtr\,(5, 1))) \\ &= pred\,(pred\,(pred\,(subtr\,(5, 0)))) \\ &= pred\,(pred\,(pred\,(5))) \\ &= pred\,(pred\,(4)) \\ &= pred\,(3) \\ &= 2. \end{aligned}$$

The Composition and Primitive Recursion Operations

♣**The composition/substitution operation**

- $h(x_1, ..., x_n) = f(g_1(x_1, ..., x_n), ..., g_m(x_1, ..., x_n))$ **(Cn)**
- $h = \mathrm{Cn}[f, g_1, ..., g_m]$, $h(x^{\rightarrow}) = \mathrm{Cn}[f, g_1, ..., g_m](x^{\rightarrow})$

♣**The primitive recursion operation**

- $h(x_1, ..., x_n, 0) = f(x_1, ..., x_n)$,
  $h(x_1, ..., x_n, y') = g(x_1, ..., x_n, y, h(x_1, ..., x_n, y))$ **(Pr)**
- $h = \mathrm{Pr}[f, g]$, $h(x^{\rightarrow}) = \mathrm{Pr}[f, g](x^{\rightarrow})$

♣**Abbreviation**

- $x^{\rightarrow} = x_1, \ldots, x_k$

Primitive Recursive Functions: Examples [BBJ-ToC-07]

6.2 Example (The addition or sum function). We start with the definition given by the equations we had above,

$$x + 0 = x \qquad x + y' = (x + y)'.$$

As a step toward reducing this to the boxed format (Pr) for recursion, we replace the ordinary plus sign, written between the arguments, by a sign written out front:

$$\mathrm{sum}(x, 0) = x \qquad \mathrm{sum}(x, y') = \mathrm{sum}(x, y)'.$$

To put these equations in the boxed format (Pr), we must find functions f and g for which we have

$$f(x) = x \qquad g(x, y, —) = s(—)$$

for all natural numbers x, y, and —. Such functions lie ready to hand: $f = \mathrm{id}^1_1$, $g = \mathrm{Cn}[s, \mathrm{id}^3_3]$. In the boxed format we have

$$\mathrm{sum}(x, 0) = \mathrm{id}^1_1(x) \qquad \mathrm{sum}(x, s(y)) = \mathrm{Cn}[s, \mathrm{id}^3_3]\,(x, y, \mathrm{sum}(x, y))$$

and in shorthand we have

$$\mathrm{sum} = \mathrm{Pr}[\mathrm{id}^1_1, \mathrm{Cn}[s, \mathrm{id}^3_3]].$$

Primitive Recursive Functions: Examples [BBJ-ToC-07]

例 6.2 （加法或者和函数）从前面给出的定义加法的等式出发，

$$x+0 = x \qquad x+y' = (x+y)'$$

作为将它们改写成方框中的递归格式（Pr）的第一步，用 sum 代替位于自变量之间的普通加号（+），并将其写在自变量的前面：

$$\mathrm{sum}(x, 0) = x \qquad \mathrm{sum}(x, y') = \mathrm{sum}(x, y)'$$

为了把这些等式改写成方框中的（Pr）格式，必须找出函数 f 和 g，为此有：

$$f(x) = x \qquad g(x, y, —) = s(—)$$

对于所有自然数 x, y 和—都成立。这样的函数随手可得：$f = \mathrm{id}^1_1$, $g = \mathrm{Cn}[s, \mathrm{id}^3_3]$，表示成方框中的格式为：

$$\mathrm{sum}(x, 0) = \mathrm{id}^1_1(x) \quad \mathrm{sum}(x, s(y)) = \mathrm{Cn}[s, \mathrm{id}^3_3](x, y, \mathrm{sum}(x, y))$$

简记为：

$$\mathrm{sum} = \mathrm{Pr}[\mathrm{id}^1_1, \mathrm{Cn}[s, \mathrm{id}^3_3]]$$

The Composition and Primitive Recursion Operations

♣**The composition/substitution operation**

- ◆$h(x_1, ..., x_n) = f(g_1(x_1, ..., x_n), ..., g_m(x_1, ..., x_n))$ (Cn)
- ◆$h = \mathrm{Cn}[f, g_1, ..., g_m]$, $h(x^{\rightarrow}) = \mathrm{Cn}[f, g_1, ..., g_m](x^{\rightarrow})$

♣**The primitive recursion operation**

- ◆$h(x_1, ..., x_n, 0) = f(x_1, ..., x_n)$,
  $h(x_1, ..., x_n, y') = g(x_1, ..., x_n, y, h(x_1, ..., x_n, y))$ (Pr)
- ◆$h = \mathrm{Pr}[f, g]$, $h(x^{\rightarrow}) = \mathrm{Pr}[f, g](x^{\rightarrow})$

♣**Abbreviation**

- ◆$x^{\rightarrow} = x_1, \ldots, x_k$

Primitive Recursive Functions: Examples [BBJ-ToC-07]

6.3 Example (The multiplication or product function). We claim prod = Pr[z, Cn[sum, id_1^3, id_3^3]]. To verify this claim we relate it to the boxed formats (Cn) and (Pr). In terms of (Pr) the claim is that the equations

$$\mathrm{prod}(x, 0) = z(x) \qquad \mathrm{prod}(x, s(y)) = g(x, y, \mathrm{prod}(x, y))$$

hold for all natural numbers x and y, where [setting $h = g$, $f = \mathrm{sum}$, $g_1 = \mathrm{id}_1^3$, $g_2 = \mathrm{id}_3^3$ in the boxed (Cn) format] we have

$$\begin{aligned} g(x_1, x_2, x_3) &= \mathrm{Cn}[\mathrm{sum}, \mathrm{id}_1^3, \mathrm{id}_3^3](x_1, x_2, x_3) \\ &= \mathrm{sum}(\mathrm{id}_1^3(x_1, x_2, x_3), \mathrm{id}_3^3(x_1, x_2, x_3)) \\ &= x_1 + x_3 \end{aligned}$$

for all natural numbers x_1, x_2, x_3. Overall, then, the claim is that the equations

$$\mathrm{prod}(x, 0) = z(x) \qquad \mathrm{prod}(x, s(y)) = x + \mathrm{prod}(x, y)$$

hold for all natural numbers x and y, which is true:

$$x \cdot 0 = 0 \qquad x \cdot y' = x + x \cdot y.$$

Primitive Recursive Functions: Examples [BBJ-ToC-07]

例 6.3 （乘法或者积函数）我们断言 prod = Pr[z, Cn[sum, id_1^3, id_3^3]]。为了验证这个断言，将它与方框（Cn）和（Pr）中的格式相联系。采用（Pr）的格式，该断言可表示为等式：

$$\mathrm{prod}(x, 0) = z(x) \qquad \mathrm{prod}(x, s(y)) = g(x, y, \mathrm{prod}(x, y))$$

对于所有自然数 x 和 y 都成立[在方框（Cn）的格式中令 $h = g$，$f = \mathrm{sum}$，$g_1 = \mathrm{id}_1^3$，$g_2 = \mathrm{id}_3^3$]，于是有：

$$\begin{aligned} g(x_1, x_2, x_3) &= \mathrm{Cn}[\mathrm{sum}, \mathrm{id}_1^3, \mathrm{id}_3^3](x_1, x_2, x_3) \\ &= \mathrm{sum}(\mathrm{id}_1^3(x_1, x_2, x_3), \mathrm{id}_3^3(x_1, x_2, x_3)) \\ &= x_1 + x_3 \end{aligned}$$

对于所有自然数 x_1, x_2, x_3 都成立。整个断言就是，等式

$$\mathrm{prod}(x, 0) = z(x) \qquad \mathrm{prod}(x, s(y)) = x + \mathrm{prod}(x, y)$$

对于所有自然数 x 和 y 都成立，即

$$x \cdot 0 = 0 \qquad x \cdot y' = x + x \cdot y$$

为真。

The Composition and Primitive Recursion Operations

♣**The composition/substitution operation**

- ◆$h(x_1, ..., x_n) = f(g_1(x_1, ..., x_n), ..., g_m(x_1, ..., x_n))$ (Cn)
- ◆$h = \mathrm{Cn}[f, g_1, ..., g_m]$, $h(x^{\rightarrow}) = \mathrm{Cn}[f, g_1, ..., g_m](x^{\rightarrow})$

♣**The primitive recursion operation**

- ◆$h(x_1, ..., x_n, 0) = f(x_1, ..., x_n)$,
  $h(x_1, ..., x_n, y') = g(x_1, ..., x_n, y, h(x_1, ..., x_n, y))$ (Pr)
- ◆$h = \mathrm{Pr}[f, g]$, $h(x^{\rightarrow}) = \mathrm{Pr}[f, g](x^{\rightarrow})$

♣**Abbreviation**

- ◆$x^{\rightarrow} = x_1, \ldots, x_k$

Primitive Recursive Functions: Examples [BBJ-ToC-07]

6.4 Example (The factorial function). The factorial $x!$ for positive x is the product $1 \cdot 2 \cdot 3 \cdot \cdots \cdot x$ of all the positive integers up to and including x, and by convention $0! = 1$. Thus we have

$$0! = 1$$
$$y'! = y! \cdot y'.$$

To show this function is recursive we would seem to need a version of the format for recursion with $n = 0$. Actually, however, we can simply define a two-argument function with a *dummy* argument, and then get rid of the dummy argument afterwards by composing with an identity function. For example, in the case of the factorial function we can define

$$\mathrm{dummyfac}(x, 0) = \mathrm{const}_1(x)$$
$$\mathrm{dummyfac}(x, y') = \mathrm{dummyfac}(x, y) \cdot y'$$

so that $\mathrm{dummyfac}(x, y) = y!$ regardless of the value of x, and then define $\mathrm{fac}(y) = \mathrm{dummyfac}(y, y)$. More formally,

$$\mathrm{fac} = \mathrm{Cn}[\mathrm{Pr}[\mathrm{const}_1, \mathrm{Cn}[\mathrm{prod}, \mathrm{id}_3^3, \mathrm{Cn}[\mathrm{s}, \mathrm{id}_2^3]]], \mathrm{id}, \mathrm{id}].$$

(We leave to the reader the verification of this fact, as well as the conversions of informal-style definitions into formal-style definitions in subsequent examples.)

Primitive Recursive Functions: Examples [BBJ-ToC-07]

例 6.4 （阶乘函数）正整数 x 的阶乘 $x!$ 是直到 x 并包括 x 的所有正整数的乘积 $1 \cdot 2 \cdot 3 \cdot \cdots \cdot x$，并约定 $0! = 1$。因此

$$0! = 1$$
$$y'! = y! \cdot y'$$

为了证明这个函数是递归的，似乎需要使用 $n = 0$ 时的递归格式。实际上，能够定义一个带哑变量的二元函数，然后通过与恒同函数的复合摆脱哑变量。例如，对于阶乘函数，可以定义：

$$\mathrm{dummyfac}(x, 0) = \mathrm{const}_1(x)$$
$$\mathrm{dummyfac}(x, y') = \mathrm{dummyfac}(x, y) \cdot y'$$

因此，不管 x 的值是什么，总是有 $\mathrm{dummyfac}(x, y) = y!$，然后定义 $\mathrm{fac}(y) = \mathrm{dummyfac}(y, y)$。更形式化的定义是，

$$\mathrm{fac} = \mathrm{Cn}[\mathrm{Pr}[\mathrm{const}_1, \mathrm{Cn}[\mathrm{prod}, \mathrm{id}_3^3, \mathrm{Cn}[s, \mathrm{id}_2^3]]], \mathrm{id}, \mathrm{id}]$$

（把验证这个形式定义以及随后例子中将非形式定义转化为形式定义的工作留给读者。）

The Composition and Primitive Recursion Operations

♣**The composition/substitution operation**

- ◆$h(x_1, ..., x_n) = f(g_1(x_1, ..., x_n), ..., g_m(x_1, ..., x_n))$ **(Cn)**
- ◆$h = \mathrm{Cn}[f, g_1, ..., g_m]$, $h(x^{\rightarrow}) = \mathrm{Cn}[f, g_1, ..., g_m](x^{\rightarrow})$

♣**The primitive recursion operation**

- ◆$h(x_1, ..., x_n, 0) = f(x_1, ..., x_n)$,
  $h(x_1, ..., x_n, y') = g(x_1, ..., x_n, y, h(x_1, ..., x_n, y))$ **(Pr)**
- ◆$h = \mathrm{Pr}[f, g]$, $h(x^{\rightarrow}) = \mathrm{Pr}[f, g](x^{\rightarrow})$

♣**Abbreviation**

- ◆$x^{\rightarrow} = x_1, \ldots, x_k$

Primitive Recursive Functions: Examples [BBJ-ToC-07]

6.5 Proposition. Let f be a primitive recursive function. Then the functions

$$g(x, y) = f(x, 0) + f(x, 1) + \cdots + f(x, y) = \sum_{i=0}^{y} f(x, i)$$

$$h(x, y) = f(x, 0) \cdot f(x, 1) \cdot \cdots \cdot f(x, y) = \prod_{i=0}^{y} f(x, i)$$

are primitive recursive.

Proof: We have for the g the recursion equations

$$g(x, 0) = f(x, 0)$$
$$g(x, y') = g(x, y) + f(x, y')$$

and similarly for h.

Primitive Recursive Functions: Examples [BBJ-ToC-07]

命题 6.5 设 f 是原始递归函数，则函数

$$g(x, y) = f(x, 0) + f(x, 1) + \cdots + f(x, y) = \sum_{i=0}^{y} f(x, i)$$

$$h(x, y) = f(x, 0) \cdot f(x, 1) \cdot \cdots \cdot f(x, y) = \prod_{i=0}^{y} f(x, i)$$

是原始递归的。

证明 定义 g 的递归等式为：

$$g(x, 0) = f(x, 0)$$
$$g(x, y') = g(x, y) + f(x, y')$$

对于 h 也有类似的递归等式。

The Composition and Primitive Recursion Operations

♣**The composition/substitution operation**

- ◆$h(x_1, ..., x_n) = f(g_1(x_1, ..., x_n), ..., g_m(x_1, ..., x_n))$ **(Cn)**
- ◆$h = \mathrm{Cn}[f, g_1, ..., g_m]$, $h(x^{\rightarrow}) = \mathrm{Cn}[f, g_1, ..., g_m](x^{\rightarrow})$

♣**The primitive recursion operation**

- ◆$h(x_1, ..., x_n, 0) = f(x_1, ..., x_n)$,
  $h(x_1, ..., x_n, y') = g(x_1, ..., x_n, y, h(x_1, ..., x_n, y))$ **(Pr)**
- ◆$h = \mathrm{Pr}[f, g]$, $h(x^{\rightarrow}) = \mathrm{Pr}[f, g](x^{\rightarrow})$

♣**Abbreviation**

- ◆$x^{\rightarrow} = x_1, \ldots, x_k$

A Notation for Super Exponentiation [BBJ-ToC-07]

Evidently the next item in the series, *super-exponentiation*, would be defined as follows:

$$x^{x^{x^{\cdot^{\cdot^{\cdot}}}}}$$ (a stack of y xs).

The alternative notation $x \uparrow y$ may be used for exponentiation to avoid piling up of superscripts. In this notation the definition would be written as follows:

$$x \uparrow x \uparrow x \uparrow \ldots \uparrow x$$ (a row of y xs).

Actually, we need to indicate the grouping here. It is to the right, like this:

$$x \uparrow (x \uparrow (x \uparrow \ldots \uparrow x \ldots))$$

and not to the left, like this:

$$(\ldots((x \uparrow x) \uparrow x) \uparrow \ldots) \uparrow x.$$

For it makes a difference: $3 \uparrow (3 \uparrow 3) = 3 \uparrow (27) = 7\,625\,597\,484\,987$; while $(3 \uparrow 3) \uparrow 3 = 27 \uparrow 3 = 19\,683$. Writing $x \Uparrow y$ for the super-exponential, the equations would be

$$x \Uparrow 0 = 0' \qquad x \Uparrow y' = x \uparrow (x \Uparrow y).$$

The next item in the series, *super-duper-exponentiation*, is analogously defined, and so on.

Primitive Recursive Functions: Examples [BBJ-ToC-07]

6.6 Example. The exponential or power function.

6.7 Example (The (modified) predecessor function). Define pred (x) to be the predecessor $x - 1$ of x for $x > 0$, and let $\mathrm{pred}(0) = 0$ by convention. Then the function pred is primitive recursive.

6.8 Example (The (modified) difference function). Define $x \dot{-} y$ to be the difference $x - y$ if $x \geq y$, and let $x \dot{-} y = 0$ by convention otherwise. Then the function $\dot{-}$ is primitive recursive.

6.9 Example (The signum functions). Define $\mathrm{sg}(0) = 0$, and $\mathrm{sg}(x) = 1$ if $x > 0$, and define $\overline{\mathrm{sg}}(0) = 1$ and $\overline{\mathrm{sg}}(x) = 0$ if $x > 0$. Then sg and $\overline{\mathrm{sg}}$ are primitive recursive.

Proofs

Example 6.6. $x \uparrow 0 = 1$, $x \uparrow s(y) = x \cdot (x \uparrow y)$, or more formally,

$$\exp = \mathrm{Pr}[\mathrm{Cn}[s, z], \mathrm{Cn}[\mathrm{prod}, \mathrm{id}_1^3, \mathrm{id}_3^3]].$$

Example 6.7. $\mathrm{pred}(0) = 0$, $\mathrm{pred}(y') = y$.
Example 6.8. $x \dot{-} 0 = x$, $x \dot{-} y' = \mathrm{pred}(x \dot{-} y)$.
Example 6.9. $\mathrm{sg}(y) = 1 \dot{-} (1 \dot{-} y)$, $\overline{\mathrm{sg}}(y) = 1 \dot{-} y$.

Primitive Recursive Functions: Examples [BBJ-ToC-07]

例 6.6 指数或者幂函数。

例 6.7 [（修改的）前趋函数]如果 $x>0$，定义 pred(x)是 x 的前趋 $x-1$，并约定 $\text{pred}(0)=0$，则函数 pred 是原始递归的。

例 6.8 [（修改的）差函数]如果 $x\geq y$，定义 $x \dot{-} y$ 是差 $x-y$，否则约定 $x \dot{-} y = 0$，则函数 $\dot{-}$ 是原始递归的。

例 6.9（数符函数）定义 $\text{sg}(0)=0$，如果 $x>0$，定义 $\text{sg}(x)=1$，并定义 $\overline{\text{sg}}(0)=1$，如果 $x>0$，定义 $\overline{\text{sg}}(x)=0$，则函数 sg 和 $\overline{\text{sg}}$ 都是原始递归的。

证明

例 6.6 $x \uparrow 0 = 1$，$x \uparrow s(y) = x \cdot (x \uparrow y)$，或可以更形式地定义为，

$$\text{exp} = \text{Pr}[\text{Cn}[s, z], \text{Cn}[\text{prod}, \text{id}^3_1, \text{id}^3_3]]$$

例 6.7 $\text{pred}(0) = 0$，$\text{pred}(y') = y$。

例 6.8 $x \dot{-} 0 = x$, $x \dot{-} y' = \text{pred}(x \dot{-} y)$。

例 6.9 $\text{sg}(y) = 1 \dot{-} (1 \dot{-} y)$, $\overline{\text{sg}}(y) = 1 \dot{-} y$。

Primitive Recursive Functions: Examples [SEP]

Constant functions

For each $k \in \mathbb{N}$ the constant k-function defined as $const_k(x) = k$ is primitive recursive. This is because we can inductively define

$$const_0(x) = \mathbf{0}$$
and
$$const_{k+1}(x) = Comp^1_1[s, const_k]$$
$$= s(s(s \ldots (s(\mathbf{0}))))$$
$$=_{\text{df}} \overline{k+1}$$

Primitive Recursive Functions: Examples [SEP]

Exponentiation, super-exponentiation, ...

We have already seen that the addition function $add(x, y)$ can be defined by primitive recursion in terms of repeated application of successor and that the multiplication function $mult(x, y)$ can be defined by primitive recursion in terms of repeated application of addition. We can continue this sequence by observing that the exponentiation function x^y can be defined by primitive recursion in terms of repeated multiplication as follows:

$$exp(x, 0) = \overline{1} \tag{20}$$
$$exp(x+1, y) = mult(x, exp(x, y))$$

The super-exponentiation function

$$x{\uparrow}y = \underbrace{x^{x^{\cdot^{\cdot^{\cdot^{x}}}}}}_{y \text{ times}}$$

can be defined by primitive recursion in terms of repeated exponentiation as as follows:

$$supexp(x, 0) = \overline{1} \tag{21}$$
$$supexp(x+1, y) = exp(x, supexp(x, y))$$

Primitive Recursive Functions: Examples [SEP]

The sequence of functions

$$\alpha_0(x, y) = x + y,$$
$$\alpha_1(x, y) = x \times y,$$
$$\alpha_2(x, y) = x^y,$$
$$\alpha_3(x, y) = x{\uparrow}y,$$
$$\vdots$$

whose $i+1$st member is defined in terms of primitive recursion of the ith member form a hierarchy of functions whose values grow increasingly quickly in proportion to their inputs. While each function in this sequence is primitive recursive, we can also consider the function $\alpha(x, y)$ defined as $\alpha_x(y, y)$—a version of the so-called *Ackermann-Péter function* defined in Section 1.4—whose values are not bounded by any fixed function α_i. As it can be shown that the values of $\alpha(x, y)$ are not bounded by any of the functions $\alpha_i(x, y)$, this shows that $\alpha(x, y)$ cannot be defined by any finite number of applications of the scheme $PrimRec_1$. This provides a constructive proof that there exist functions of type $\mathbb{N}^2 \to \mathbb{N}$ which are not primitive recursive.

Primitive Recursive Functions: Examples [SEP]

Predecessor and proper subtraction

The *proper predecessor* function is given by

$$pred(y) = \begin{cases} 0 & \text{if } y = 0 \\ y - 1 & \text{otherwise} \end{cases}$$

This function is primitive recursive since it may be defined as

$$pred(y) = 0 \tag{22}$$
$$pred(y+1) = y$$

Note that the second clause of (22) does not depend on the prior value of $pred(y)$. But this definition can still be conformed to the scheme (18) by taking $f(x_0) = \mathbf{0}$ and $g(x_0, x_1, x_2) = \pi^1_1$.

The *proper subtraction* function is given by

$$x \dot{-} y = \begin{cases} x - y & \text{if } y \leq x \\ 0 & \text{otherwise} \end{cases}$$

This function is also primitive recursive since it may be defined as

$$x \dot{-} 0 = x \tag{23}$$
$$x \dot{-} (y+1) = pred(x \dot{-} y)$$

Primitive Recursive Functions: Examples [SEP]

Absolute difference, signum, minimum, and maximum

The absolute difference function is defined as

$$|x - y| = \begin{cases} x - y & \text{if } y \leq x \\ y - x & \text{otherwise} \end{cases}$$

$|x - y|$ may be defined by composition as $(x \dot{-} y) + (y \dot{-} x)$ and is hence primitive recursive since $\dot{-}$ is.

The *signum* function is defined as

$$sg(x) = \begin{cases} 1 & \text{if } x \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

This function may be defined by composition as $sg(x) = 1 \dot{-} (1 \dot{-} x)$ and is hence primitive recursive as is the *inverted signum* function defined by $\overline{sg}(x) = 1 \dot{-} sg(y)$ which returns 1 if $x = 0$ and 1 otherwise.

The minimum and maximum functions may be similarly defined by composition from functions previously seen to be primitive recursive as follows:

$$\min(x, y) = \overline{sg}(x \dot{-} y) \times x + \overline{sg}(y \dot{-} x) \times y$$
$$\max(x, y) = sg(x \dot{-} y) \times x + sg(y \dot{-} x) \times y$$

Primitive Recursive Functions: Examples [SEP]

Order and identity

The characteristic functions of the *less than* relation ($<$) and *equality* relation ($=$) on the natural numbers are definable as follows:

$$\chi_{<}(x, y) = sg(y \dot{-} x)$$
$$\chi_{=}(x, y) = 1 \dot{-} (sg(x \dot{-} y) + sg(y \dot{-} x))$$

These relations are hence primitive recursive.

As the *less than or equal to* relation ($\leq$) is logically equivalent to $x < y \vee x = y$ it will follow from the next set of observations that this relation is also primitive recursive. The is additionally true of $x > y$, $x \geq y$ and $x \neq y$.

Closure under propositional operations

The set of primitive recursive relations is *closed under boolean operations*. In other words, if $P(\vec{x})$ and $Q(\vec{x})$ are primitive recursive, then so are $\neg P(\vec{x})$, $P(\vec{x}) \wedge Q(\vec{x})$, $P(\vec{x}) \vee Q(\vec{x})$, $P(\vec{x}) \rightarrow Q(\vec{x})$, and $P(\vec{x}) \leftrightarrow Q(\vec{x})$.

Given the interdefinability of the classical connectives, this follows upon noting the following:

$$\chi_{\neg P}(\vec{x}) = 1 \dot{-} \chi_P(\vec{x})$$
$$\chi_{P \wedge Q}(\vec{x}) = \chi_P(\vec{x}) \times \chi_Q(\vec{x})$$

Primitive Recursive Functions: Examples [SEP]

Bounded sums and products

Suppose that $f(\vec{x}, z)$ is primitive recursive. Then the *bounded sum* $g(\vec{x}, y) = \Sigma_{i=0}^{y} f(\vec{x}, i)$ and the *bounded product* $h(\vec{x}, y) = \Pi_{i=0}^{y} f(\vec{x}, i)$ are both primitive recursive as they may be respectively defined as follows:

$$g(\vec{x}, 0) = f(\vec{x}, 0)$$
$$g(\vec{x}, y + 1) = g(\vec{x}, y) + f(\vec{x}, y + 1)$$
$$h(\vec{x}, 0) = f(\vec{x}, 0)$$
$$h(\vec{x}, y + 1) = g(\vec{x}, y) \times f(\vec{x}, y + 1)$$

Closure under bounded quantification

The set of primitive recursive relations is also closed under *bounded quantification*—i.e., if $R(\vec{x}, z)$ is a primitive recursive relation, then so are the relations $\forall z \leq y R(\vec{x}, z)$ and $\exists z \leq y R(\vec{x}, z)$. These may be respectively defined as follows as follows:

$$u_R(\vec{x}, y) =_{\text{df}} \chi_{\forall z \leq y R(\vec{x}, z)}(\vec{x}) = \Pi_{i=0}^{y} \chi_R(\vec{x}, i)$$
$$e_R(\vec{x}, y) =_{\text{df}} \chi_{\exists z \leq R(\vec{x}, z)}(\vec{x}) = sg\left(\Sigma_{i=0}^{y} \chi_R(\vec{x}, i)\right)$$

Primitive Recursive Functions: Examples [SEP]

Closure under bounded minimization

The set of primitive recursive relations is also closed under *bounded minimization*. This is to say that if $R(\vec{x}, z)$ is a primitive recursive relation, then so is the function $m_R(\vec{x}, y)$ which returns the least z less than or equal to y such that $R(\vec{x}, z)$ holds if such a z exists and $y + 1$ otherwise—i.e.,

$$(24) \qquad m_R(\vec{x}, y) = \begin{cases} \text{the least } z \leq y \text{ such that } R(\vec{x}, z) & \text{if such a } z \text{ exists} \\ y + 1 & \text{otherwise} \end{cases}$$

To see this, observe that if $R(\vec{x}, z)$ is primitive recursive, then so is $\forall z \leq y \neg R(\vec{x}, z)$. It is then not difficult to verify that

$$m_R(\vec{x}, y) = \Sigma_{i=0}^{y} \chi_{\forall z \leq y \neg R(\vec{x}, z)}(\vec{x}, i).$$

Primitive Recursive Functions: Examples [SEP]

Divisibility and primality

A natural number y is said to be *divisible* by x just in case there exists a z such that $x \times z = y$—i.e., x divides y without remainder. In this case we write $x \mid y$. Note that if $x \mid y$ holds, then this must be witnessed by a divisor $z \leq y$ such that $x \times z = y$. We may thus define $x \mid y$ in the following manner which shows that it is primitive recursive:

$$x \mid y \iff \exists z \leq y (x \times z = y)$$

We may also define the *non-divisibility* relations $x \nmid y$ as $\neg(x \mid y)$ which shows that it too is primitive recursive.

Next recall that a natural number x is *prime* just in case it is greater than 1 and is divisible by only 1 and itself. We may thus define the relation $Prime(x)$ in the following manner which shows that it is primitive recursive:

$$Prime(x) \iff \overline{1} < x \wedge \forall z \leq x (z \mid x \rightarrow (z = \overline{1} \vee z = x))$$

The primes form a familiar infinite sequence $p_0 = 2, p_1 = 3, p_2 = 5, p_3 = 7, p_4 = 11, \ldots$ Let $p(x) = p_x$—i.e., the function which returns the xth prime number. $p(x)$ can be defined by primitive recursion relative to the function $nextPrime(x)$ which returns the least $y > x$ such that y is prime as follows:

$$p(0) = \overline{2}$$
$$p(x + 1) = nextPrime(p(x))$$

Recall that Euclid's Theorem states that there is always a prime number between x and $x! + 1$ and also that $x! = fact(x)$ is primitive recursive. It thus follows that $nextPrime(x)$ can be defined via bounded minimization as follows:

$$nextPrime(x) = m_{x<z \wedge Prime(z)}(x, fact(x) + 1)$$

It thus follows that $p(x)$ is primitive recursive.

Primitive Recursive Functions: Expressive Power

♣ **The expressive power of primitive recursive functions**

- **The expressive power of primitive recursive functions is considerable, and most common functions are primitive recursive.**
- **However, not all functions are in this class, there are some function that are not primitive recursive.**

Primitive Recursive Functions: Expressive Power [L-ToC-17]

♣ **The expressive power of primitive recursive functions**

THEOREM 13.1

Let F denote the set of all functions from I to I. Then there is some function in F that is not primitive recursive.

Proof: Every primitive recursive function can be described by a finite string that indicates how it is defined. Such strings can be encoded and arranged in standard order. Therefore, the set of all primitive recursive functions is countable.

Suppose now that the set of all functions is also countable. We can then write all functions in some order, say, $f_1, f_2, \ldots$. We next construct a function g defined as

$$g(i) = f_i(i) + 1, \qquad i = 1, 2, \ldots.$$

Clearly, g is well defined and is therefore in F, but equally clearly, g differs from every f_i in the diagonal position. This contradiction proves that F cannot be countable.

Combining these two observations proves that there must be some function in F that is not primitive recursive. ■

Primitive Recursive Functions: Expressive Power [L-ToC-17]

♣ **The expressive power of primitive recursive functions**

THEOREM 13.2

Let C be the set of all total computable functions from I to I. Then there is some function in C that is not primitive recursive.

Proof: By the argument of the previous theorem, the set of all primitive recursive functions is countable. Let us denote the functions in this set as $r_1, r_2, \ldots$ and define a function g by

$$g(i) = r_i(i) + 1.$$

By construction, the function g differs from every r_i and is therefore not primitive recursive. But clearly g is computable, proving the theorem. ∎

♣ **Note**

◆ **"Computable" is not formally defined here.**

Ackermann's Function [L-ToC-17]

♣ **The definition of Ackermann's function**

Ackermann's Function

Ackermann's function is a function from $I \times I$ to I, defined by

$$\begin{aligned} A(0, y) &= y + 1, \\ A(x, 0) &= A(x - 1, 1), \\ A(x, y + 1) &= A(x - 1, A(x, y)). \end{aligned}$$

♣ **Fact**

◆ **Ackermann's function is not primitive recursive.**

Ackermann's Function [L-ToC-17]

♣ **The limit of growth for primitive recursive functions**

THEOREM 13.3

Let f be any primitive recursive function. Then there exists some integer n such that

$$f(i) < A(n, i),$$

for all $i = n, n + 1, \ldots$.

♣ **Ackermann's function is not primitive recursive**

THEOREM 13.4

Ackermann's function is not primitive recursive.

Proof: Consider the function

$$g(i) = A(i, i).$$

If A were primitive recursive, then so would g. But then, according to Theorem 13.3, there exists an n such that

$$g(i) < A(n, i),$$

for all i. If we now pick $i = n$, we get the contradiction

$$\begin{aligned} g(n) &= A(n, n) \\ &< A(n, n), \end{aligned}$$

proving that A cannot be primitive recursive. ∎

μ Recursive Functions [L-ToC-17]

♣ **μ (minimalization) operator**

◆ **Let *g* be a total function. The *μ operator* or *minimalization operator*, defined by:**
μy(*g*(*x*, *y*)) = smallest *y* such that *g*(*x*, *y*) = 0.

♣ **The definition of μ recursive functions**

DEFINITION 13.2

A function is said to be μ-recursive if it can be constructed from the basis functions by a sequence of applications of the μ-operator and the operations of composition and primitive recursion.

THEOREM 13.5

A function is μ-recursive if and only if it is computable.

♣ **Note**

◆ **"Computable" is not formally defined here.**

μ Recursive Functions: An Example [L-ToC-17]

EXAMPLE 13.4

Let

$$g(x, y) = x + y \dot{-} 3,$$

which is a total function. If $x \leq 3$, then

$$y = 3 - x$$

is the result of the minimalization, but if $x > 3$, then there is no $y \in I$ such that $x + y - 3 = 0$. Therefore,

$$\begin{aligned} \mu y(g(x, y)) &= 3 - x, \qquad \text{for } x \leq 3, \\ &= \text{undefined, for } x > 3. \end{aligned}$$

We see from this that even though $g(x, y)$ is a total function, $\mu y(g(x, y))$ may only be partial.

Effectively Computable Partial Functions [BBJ-ToC-07]

♣ **Effectively computable partial functions**

◆ **Intuitively, we consider a partial function *f* to be *effectively computable* if a list of definite, explicit instructions can be given, following which one will, in the case they are applied to any *x* in the domain of *f*, arrive after a finite number of steps at the value *f*(*x*), but following which one will, in the case they are applied to any *x* not in the domain of *f*, go on forever without arriving at any result.**

◆ **This notion applies also to two- and many-place functions.**

Minimization [BBJ-ToC-07]

♣ **Minimization operation**

- Given a total or partial function f of n+1 arguments, the operation of ***minimization*** yields a total or partial function h of n arguments as follows:

$$\mathrm{Mn}[f](x_1,\ldots,x_n) = \begin{cases} y & \text{if } f(x_1,\ldots,x_n,y)=0, \text{ and for all } t<y \\ & \quad f(x_1,\ldots,x_n,t) \text{ is defined and} \neq 0 \\ \text{undefined} & \text{if there is no such } y. \end{cases}$$

♣ **The simpler form**

- In case f is a total function, the above definition boils down to the following simpler form:

$$\mathrm{Mn}[f](x_1,\ldots,x_n) = \begin{cases} \text{the smallest } y \text{ for which} & \\ \quad f(x_1,\ldots,x_n,y)=0 & \text{if such a } y \text{ exists} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Minimization [BBJ-ToC-07]

♣ **Effectively computable function**

- If h = Mn[f] and f is an effectively computable total or partial function, then h also will be such a function.

♣ **The computation steps of h**

- For writing x for x_1, ..., x_n, we compute $h(x)$ by successively computing $f(x, 0)$, $f(x, 1)$, $f(x, 2)$, and so on, stopping if and when we reach a y with $f(x, y) = 0$.
- If x is in the domain of h, there will be such a y, and the number of steps needed to compute $h(x)$ will be the sum of the number of steps needed to compute $f(x,0)$, the number of steps needed to compute $f(x, 1)$, and so on, up through the number of steps needed to compute $f(x, y) = 0$.
- If x is not in the domain of h, this may be for either of two reasons. On the one hand, it may be that all of $f(x, 0)$, $f(x, 1)$, $f(x, 2)$, ... are defined, but they are all nonzero. On the other hand, it may be that for some i, all of $f(x, 0)$, $f(x, 1)$, ..., $f(x, i-1)$ are defined and nonzero, but $f(x, i)$ is undefined. In either case, the attempt to compute $h(x)$ will involve one in a process that goes on forever without producing a result.

Minimization [BBJ-ToC-07]

♣ **Regular functions**

- The total function f is called ***regular*** if for every x_1, ..., x_n there is a y such that $f(x_1, ..., x_n, y) = 0$.

♣ **Facts**

- In case f is a regular function, Mn[f] will be a total function.
- In fact, if f is a total function, Mn[f] will be total if and only if f is regular.

Partial Recursive Functions [W-ToC-12]

Definition 3.3.9. The class of *partial recursive functions* is the smallest class of functions containing (i), (ii), and (iii) from Definition 3.3.1 of the primitive recursive functions, and closed under (iv) and (v) from that definition as well as

(vi) (Unbounded search, minimization, or μ-recursion.) If $\bar{x} = x_1, \ldots, x_n$, $\theta(\bar{x}, y)$ is a partial recursive function of $n+1$ variables, and we define $\psi(\bar{x})$ to be the least y such that $\theta(\bar{x}, y) = 0$ and $\theta(\bar{x}, z)$ is defined for all $z < y$, then ψ is a partial recursive function of n variables.

Partial Recursive Predicates [F-ToC-09]

Definition 4.5 (Primitive recursive predicates)

The condition P depending on $X \in \mathsf{Nat}^n$, such that $P(X)$ is either true or false, is called an *n-ary predicate*. An n-ary predicate P is primitive recursive if its *characteristic function* $\mathcal{X}_P : \mathsf{Nat}^n \to \{0,1\}$ is primitive recursive. The characteristic function of a predicate associates 1 with the tuples X for which $P(X)$ holds and 0 with the others.

Example 4.6

The predicates eq (equality) and lt (less than) are primitive recursive with characteristic functions

$$\begin{aligned} \mathcal{X}_{\mathsf{lt}}(x,y) &= f(\mathsf{sub}(y,x)) \\ \mathcal{X}_{\mathsf{eq}}(x,y) &= f(\mathsf{add}(\mathsf{sub}(x,y),\mathsf{sub}(y,x))) \end{aligned}$$

where $f(0) = 0$ and $f(\mathsf{S}(n)) = 1$. The function f is primitive recursive (see the

(General) Recursive Functions

♣ **(General) Recursive functions**

- The functions that can be obtained from the basic functions z, s, id_i^n by the processes Cn, Pr, and Mn are called the ***(general) recursive*** (total or partial) ***functions***.

♣ **Fact**

- Recursive functions are all effectively computable.

♣ **Note**

- In the literature, 'recursive function' is often used to mean more specifically 'recursive *total* function', and 'partial recursive function' is then used to mean 'recursive total or partial function'.

Recursive Functions: μ Operation [SEP]

- A function $f : \mathbb{N}^k \to \mathbb{N}$ is called *total* if $f(\vec{n})$ is defined for all $\vec{n} \in \mathbb{N}^k$. Otherwise $f(\vec{x})$ is called *partial*.
- We write $f(\vec{n})\downarrow$ to express that $f(\vec{x})$ is defined at $\vec{n}$ and additionally $f(\vec{n})\downarrow = m$ if $f(\vec{n})$ is defined at $\vec{n}$ and equal to m. Otherwise we write $f(\vec{n})\uparrow$ to express that $f(\vec{x})$ is undefined at $\vec{n}$.
- The *domain* of $f(\vec{n})$ is the set $\mathrm{dom}(f) = \{\vec{n} \in \mathbb{N}^k : f(\vec{n})\downarrow\}$
- We write $f(\vec{x}) \simeq g(\vec{x})$ just in case for all $\vec{n} \in \mathbb{N}$, either $f(\vec{n})$ and $g(\vec{n})$ are both undefined or are both defined and equal.

Suppose we are given a partial function $f(x_0, \ldots, x_{k-1}, y)$. We now introduce terms of the form $\mu y f(x_0, \ldots, x_{k-1}, y)$ defined as follows:

$$\mu y f(x_0, \ldots, x_{k-1}, y) = \begin{cases} z & \text{if } z \text{ is such that} \\ & f(x_0, \ldots, x_{k-1}, z) = 0 \text{ and} \\ & \forall w < z (f(x_0, \ldots, x_1, w)\downarrow \neq 0) \\ \uparrow & \text{otherwise} \end{cases} \tag{28}$$

In other words, $\mu y f(\vec{n}, y)$ is equal to the least m such that $f(\vec{n}, m) = 0$ provided that such an m exists and also that $f(\vec{n}, i)$ is defined but not equal to 0 for all $0 \leq i < m$. On the other hand, $\mu y f(\vec{n}, y)$ is undefined just in case either there is no m such that $f(\vec{n}, m) = 0$ or there is such a m but $f(\vec{n}, i)$ is undefined for some $i < m$.

Partial and Total Recursive Functions (PartREC and REC) [SEP]

Since this definition determines $\mu y f(x, y)$ uniquely, (28) can also be regarded as defining a functional $\mathcal{M}in_k$ which maps $k+1$-ary partial functions into k-ary partial functions. We now define the classes of functions **PartREC** and **REC** as follow:

Definition 2.5: The class of *partial recursive functions* **PartREC** (also known as the *μ-recursive functions*) is the smallest class of partial functions of type $\mathbb{N}^k \to \mathbb{N}$ containing the initial functions $I_{\mathbf{PR}} = \{\mathbf{0}, s, \pi_k^i\}$ and closed under the functionals

$$Op_{\mathbf{PartREC}} = \{Comp_j^i, \mathcal{P}rim\mathcal{R}ec_k, \mathcal{M}in_k\}.$$

We say that a function $f : \mathbb{N}^k \to \mathbb{N}$ is *partial recursive* if $f \in$ **PartREC**. Additionally we say that f is *recursive* if $f \in$ **PartREC** and f is total. The set of recursive functions will be denoted by **REC**.

Note that despite its name, *the class of partial recursive functions contains total functions*. In particular, a *recursive function* is, by definition, one which is *partial recursive while also being total*. We will see in Section 3.2, there also exist partial recursive functions which are genuinely partial and total functions which are not recursive.

Kleene Normal Form Theorem [SEP]

A question which naturally arises at this stage is whether more than one application of unbounded minimization is required to obtain all partial recursive functions. The fact that a single application is sufficient is a consequence of the *Kleene Normal Form Theorem*. In order to formulate this result, it is convenient to officially extend the application of the μ-operator to relations in the manner discussed at the beginning of this section—i.e.,

$$\mu y R(\vec{x}, y) = \begin{cases} \text{the least } y \text{ such that } R(\vec{x}, y) & \text{if such a } y \text{ exists} \\ \uparrow & \text{otherwise} \end{cases} \tag{29}$$

Theorem 2.3: For all $k \in \mathbb{N}$ there exists a $k+2$-ary primitive recursive relation $T_k(e, \vec{x}, s)$—the so-called *Kleene T-predicate*—and a primitive recursive function $u(x)$ (not depending on k) satisfying the following condition: for all k-ary partial recursive functions $f(\vec{x})$ there exists $e \in \mathbb{N}$ such that for all $\vec{n} \in \mathbb{N}^k$

$$f(\vec{n}) \simeq u(\mu s T_k(e, \vec{n}, s))$$

Kleene Normal Form Theorem [SEP]

Since $\mu y R(\vec{x}, y) \simeq \mu y \chi_{\neg R}(\vec{x}, y)$ it is easy to see that the class **PartREC** can also be obtained by closing the primitive recursive functions under the operation defined by (29). One consequence of Theorem 2.3 is thus that it is indeed possible to define any k-ary partial recursive function $f(\vec{x})$ by a single application of unbounded search applied to the relation $T_k(e, \vec{x}, s)$ for an appropriate choice of e. More generally, the Normal Form Theorem illustrates how any such function may be defined from a *single* relation $T_k(e, \vec{x}, s)$ wherein the value of e serves as a description of the manner in which $f(\vec{x})$ is defined in terms of the basis functions $I_{\mathbf{PR}}$ and the operations $Op_{\mathbf{PartRec}}$. Such an e is known as an *index* for $f(\vec{x})$. As we will see in Section 3, the availability of such indices is one of the central features of the partial recursive functions which allows them to provide the basis for a general theory of computability and non-computability.

Recursive Functions and the Church-Turing Thesis

♣ **Hypothesis (The Church-Turing thesis)**

◆ **All effectively computable total functions are recursive.**

♣ **Hypothesis (The Church-Turing thesis): An extended version**

◆ **All effectively computable partial functions are recursive.**

♣ **Note**

◆ **At present the Church-Turing thesis is simply an hypothesis.**

Intuitive notion of algorithms	equals	*Turing machine algorithms*

FIGURE 3.22
The Church–Turing Thesis

The Computability Theory of Recursive Functions [SEP]

♣ **Kleene's proof of the Normal Form Theorem**

◆ **The first significant result in computability theory (of recursive functions) was Kleene's (1936a) proof of the Normal Form Theorem.**

◆ **The Normal Form Theorem can be understood as illustrating how it is possible to associate each k-ary partial computable function with a natural number known as its index such that $f(x^{\rightarrow}) \simeq \mu s(T_k(e, x^{\rightarrow}, s))$.**

♣ **Indexation of k-ary partial computable functions**

◆ **This also leads to what is known as an indexation of k-ary partial computable functions:**

$$\phi_0^k(\vec{x}), \phi_1^k(\vec{x}), \phi_2^k(\vec{x}), \ldots, \phi_i^k(\vec{x}), \ldots \text{ where } \phi_i^k(\vec{x}) \simeq \mu s T_k(i, \vec{x}, s)$$

◆ **Such an enumeration provides a uniform means of listing off all partial computable functions in the order of their indices.**

Consequences of the Normal Form Theorem [SEP]

♣ The s-m-n Theorem

Theorem 3.1: For all $n, m \in \mathbb{N}$, there is a primitive recursive function $s_n^m(i, x_0, \ldots, x_{m-1})$ such that

$$\phi^n_{s_n^m(i,x_0,\ldots,x_{m-1})}(y_0, \ldots, y_{n-1}) \simeq \phi_i^{n+m}(x_0, \ldots, x_{m-1}, y_0, \ldots, y_{n-1})$$

Here the function $s_n^m(i, \vec{x})$ should be thought of as acting on an index i for an $n + m$-ary partial computable function together with values $\vec{x}$ for the first m of its arguments. This function returns an index for another partial computable function which computes the n-ary function determined by carrying out ϕ_i^{n+m} with the first m of its arguments $\vec{x}$ fixed but retaining the next n variables $\vec{y}$ as inputs. Although the formulation of

♣ The k-ary universal partial computable function

Theorem 3.2: For every $k \in \mathbb{N}$, there is a $k + 1$-ary partial computable function v^k which is universal in the sense that for all k-ary partial computable functions $f(\vec{x})$, there is an $i \in \mathbb{N}$ such that $v_k(i, \vec{x}) \simeq f(\vec{x})$.

This follows immediately from Theorem 2.3 by taking $v_k(i, \vec{x}) = u(\mu s T_k(i, \vec{x}, s))$ where i is such that $f(\vec{x}) \simeq \phi_i^k(\vec{x})$ in the enumeration of k-ary partial computable functions. As $v^k(i, \vec{x})$ can be used to compute the values of all k-ary partial computable functions uniformly in their index, it is conventionally referred to as the k-ary *universal partial computable function*.

The Computability Theory of Recursive Functions [SEP]

♣ A single enumeration of unary functions

- **While we have just defined such a function for each k, it is also possible to define a binary function $v(i, x)$ which treats its second argument as a code for a finite sequence $x_0, \ldots, x_{k-1}$ and then computes in the same manner as the k-ary universal function so that we have $v(i, \langle x_0, \ldots, x_{k-1}\rangle) \simeq v^k(i, x_0, \ldots, x_{k-1})$.**
- **This provides a means of replacing the prior enumerations of k-ary partial computable functions with a single enumeration of unary functions:**

$$\phi_0(x), \phi_1(x), \phi_2(x), \ldots, \phi_i(x), \ldots$$

where $\phi_i(\langle x_0, \ldots, x_{k-1}\rangle) \simeq v(i, \langle x_0, \ldots, x_{k-1}\rangle) \simeq \phi_i^k(x_0, \ldots, x_{k-1})$

The Computability Theory of Recursive Functions [SEP]

♣ Development of a general theory of computability

- **Together with the Kleene Normal Form Theorem, the s-m-n Theorem and Theorem 3.2 codify the basic properties of a model of computation which make it suitable for the development of a general theory of computability.**

Non-Computable Functions and Undecidable Problems [SEP]

♣ The difference between partial computable functions and primitive recursive functions

- **The partial computable functions differ from the primitive recursive functions in admitting a universal function within the same class but at the same time giving up the requirement that the functions in the class must be total.**
- **In other words, while $v(i, x) \in$ PartREC, the discussion in Section 2.2.2 shows that $u_1(i, x^{\rightarrow}) \in$ REC − PR.**

Non-Computable Functions and Undecidable Problems [SEP]

♣ Halting Problem for the Turing Machine model

- **The problem was originally formulated by Turing (1937) as follows:**
  Given an indexation of T_0, T_1, … of Turing machines, does machine T_i halt on the input n?
- **An equivalent question can also be formulated in terms of the partial recursive functions:**
  Is the partial computable function $\phi_i(x)$ defined for input n?
- **The pairs of natural numbers $\langle i, n\rangle$ corresponding to positive answers to this question determine a subset of $N \times N$ as follows: HP = $\{ \langle i, n\rangle \mid \phi_i(x)\downarrow \}$**
- **Let $h(x, y)$ be $\chi_{HP}(x, y)$, by definition, this is a total function.**
- **Theorem: $h(x, y)$ is not a computable function.**

An Introduction to the Theory of Computation

- **Enumerability and Diagonalization**
- **Finite Automata and Regular Languages**
- **Pushdown Automata and Context-Free Languages**
- **Computation: Turing Machines**
- **Computation: Turing-Computability (Turing-Decidability)**
- **Computation: Reducibility and Turing-Reducibility**
- **Computation: Recursive Functions**
- **Computation: Recursive Sets and Relations**
- **Equivalent Definitions of Computability**
- **Advanced Topics in Computability Theory**
- **Computational Complexity**
- **Time Complexity**
- **Space Complexity**
- **Intractability**
- **Advanced Topics in Complexity Theory**

Some Function Definitions

♣ The difference function

- The *difference function* $-^{\bullet}$: $x -^{\bullet} y =_{df} x - y$ if $y < x$, and = 0 otherwise.

$x \dot{-} y = x - y$ if $y < x$, and $= 0$ otherwise.

♣ The signum function

- The *signum function sg*: $sg(x) =_{df} 1$ if $x > 0$, and = 0 otherwise.

♣ The graph relation

- For a given function f, the *graph relation* of f is the relation defined by $G(x_1, ..., x_n, y) \leftrightarrow f(x_1, ..., x_n) = y$.

Effectively Decidable Sets

♣ Effectively decidable sets

- A set of, say, natural numbers is *effectively decidable* if there is an effective procedure that, applied to a natural number, in a finite amount of time (steps) gives the correct answer to the question whether it belongs to the set.

♣ The characteristic functions of sets

- Let A be a set. The *characteristic function* of the set A is the function cf_A: $A \rightarrow \{1, 0\}$ that takes the value 1 for members in A, and the value 0 for members not in A.
- Representing the answer 'YES' by 1 and the answer 'NO' by 0, a set is *effectively decidable* IFF its characteristic function is effectively computable.

Recursive Sets

♣ Recursive sets and primitive recursive sets

- A set is called *recursively decidable*, or simply *recursive* for short, if its characteristic function is recursive, and is called *primitive recursive* if its characteristic function is primitive recursive.

♣ Facts about recursive sets

- Since recursive functions are effectively computable, recursive sets are effectively decidable.
- Church-Turing's thesis, according to which all effectively computable functions are recursive, implies that all effectively decidable sets are recursive.

Recursive Relations [BBJ-ToC-07]

♣ The characteristic functions of k-ary relations

- A k-ary relation R is a set of ordered k-tuples (i.e., $(x_1, ..., x_k) \in R$, we write Sx or $S(x)$ interchangeably with $x \in S$).
- The *characteristic function* of a k-ary relation is the k-argument function that takes the value 1 for a k-tuple if the relation holds of that k-tuple, and the value 0 if it does not.

♣ Recursive relations

- A relation is *effectively decidable* if its characteristic function is effectively computable, and is *(primitive) recursive* if its characteristic function is (primitive) recursive.

(Primitive) Recursive Relations: Examples [BBJ-ToC-07]

7.1 Example (Identity and order). The identity relation, which holds if and only if $x = y$, is primitive recursive, since a little thought shows its characteristic function is $1 - (sg(x \dot{-} y) + sg(y \dot{-} x))$. The strict less-than order relation, which holds if and only if $x < y$, is primitive recursive, since its characteristic function is $sg(y \dot{-} x)$.

例 7.1 （恒同关系与序关系）x 和 y 满足恒同关系当且仅当 $x=y$。恒同关系是原始递归的，因为稍微想一想就会发现，它的特征函数是 $1 \dot{-} (sg(x \dot{-} y) + sg(y \dot{-} x))$。$x$ 和 y 满足严格小于关系当且仅当 $x<y$。严格小于关系是原始递归的，因为它的特征函数是 $sg(y \dot{-} x)$。

(Primitive) Recursive Relations: Examples [BBJ-ToC-07]

7.2 Proposition (Definition by cases). Suppose that f is the function defined in the following form:

$$f(x, y) = \begin{cases} g_1(x, y) & \text{if } C_1(x, y) \\ \vdots & \vdots \\ g_n(x, y) & \text{if } C_n(x, y) \end{cases}$$

where $C_1, \ldots, C_n$ are (primitive) recursive relations that are mutually exclusive, meaning that for no x, y do more than one of them hold, and collectively exhaustive, meaning that for any x, y at least one of them holds, and where $g_1, \ldots, g_n$ are (primitive) recursive total functions. Then f is (primitive) recursive.

Proof: Let c_i be the characteristic function of C_i. By recursion, define $h_i(x, y, 0) = 0$, $h_i(x, y, z') = g_i(x, y)$. Let $k_i(x, y) = h_i(x, y, c_i(x, y))$, so $k_i(x, y) = 0$ unless $C_i(x, y)$ holds, in which case $k_i(x, y) = g_i(x, y)$. It follows that $f(x, y) = k_1(x, y) + \ldots + k_n(x, y)$, and f is (primitive) recursive since it is obtainable by primitive recursion and composition from the g_i and the c_i, which are (primitive) recursive by assumption, together with the addition (and identity) functions.

(Primitive) Recursive Relations: Examples [BBJ-ToC-07]

命题 7.2 （分情况定义）假设函数 f 定义如下：

$$f(x,y)=\begin{cases} g_1(x,y) & \text{如果} C_1(x,y) \\ \vdots & \vdots \\ g_n(x,y) & \text{如果} C_n(x,y) \end{cases}$$

其中（原始）递归关系 $C_1, \cdots, C_n$ 是互斥的，即没有 x，y 满足其中多个关系并且是集体完全的，即任意 x，y 满足其中至少一个关系，$g_1, \cdots, g_n$ 是（原始）递归全函数。那么，f 是（原始）递归的。

证明 因为 f 能由加法函数、乘法函数、函数 g_i 以及关系 C_i 的特征函数 c_i 通过如下复合得出：

$$f(x,y)=g_1(x,y)\cdot c_1(x,y)+\cdots+g_n(x,y)\cdot c_n(x,y)$$

对于作为自变量的每对自然数 x, y，特征函数 $c_1, \cdots, c_n$ 的值中只有一个不是 0，比如说 c_i 的值是 1，这时数 x, y 实际上满足对应条件 C_i，因此上面的等式成立。

(Primitive) Recursive Relations: Examples [BBJ-ToC-07]

7.3 Example (The maximum and minimum functions). As an example of definition by cases, consider $\max(x, y)$ = the larger of the numbers x, y. This can be defined as follows:

$$\max(x,y)=\begin{cases} x & \text{if } x \geq y \\ y & \text{if } x < y \end{cases}$$

or in the official format of the proposition above with $g_1 = \mathrm{id}_1^2$ and $g_2 = \mathrm{id}_2^2$. Similarly, function $\min(x, y)$ = the smaller of x, y is also primitive recursive.

例 7.3 （最大值函数与最小值函数）作为分情况定义的例子，考虑函数 $\max(x, y)$ = 数 x, y 中的较大者。这个函数可定义如下：

$$\max(x,y)=\begin{cases} x & \text{如果 } x \geqslant y \\ y & \text{如果 } x < y \end{cases}$$

若采用上面命题中的正式格式，则令 $g_1 = \mathrm{id}_1^2$ 和 $g_2 = \mathrm{id}_2^2$ 即可。类似地，函数 min 也是原始递归的，其中 $\min(x, y)$ =数 x, y 中的较小者。

(Primitive) Recursive Functions Defined by Cases

- **To establish the (primitive) recursiveness**
  - **Definition by cases makes it far easier to establish the (primitive) recursiveness of important functions.**
  - **This is mainly because there are a variety of processes for defining new relations from old that can be shown to produce new (primitive) recursive relations when applied to (primitive) recursive relations.**

(Primitive) Recursive Functions Defined by Cases [BBJ-ToC-07]

- **The result of substituting recursive total functions in a recursive relation is itself a recursive relation**

Given a relation $R(y_1, \ldots, y_m)$ and total functions $f_1(x_1, \ldots, x_n), \ldots, f_m(x_1, \ldots, x_n)$, the relation defined by *substitution* of the f_i in R is the relation $R^*(x_1, \ldots, x_n)$ that holds of $x_1, \ldots, x_n$ if and only if R holds of $f_1(x_1, \ldots, x_n), \ldots, f_m(x_1, \ldots, x_n)$, or in symbols,

$$R^*(x_1,\ldots,x_n) \leftrightarrow R(f_1(x_1,\ldots,x_n),\ldots,f_m(x_1,\ldots,x_n)).$$

If the relation R^* is thus obtained by substituting functions f_i in the relation R, then the characteristic function c^* of R^* is obtainable by composition from the f_i and the characteristic function c of R:

$$c^*(x_1,\ldots,x_n) = c(f_1(x_1,\ldots,x_n),\ldots,f_n(x_1,\ldots,x_n)).$$

Therefore, *the result of substituting recursive total functions in a recursive relation is itself a recursive relation.* (Note that it is important here that the functions be *total*.)

(Primitive) Recursive Functions Defined by Cases [BBJ-ToC-07]

- **The graph relation of a recursive total function is a recursive relation**

An illustration may make this important notion of substitution clearer. For a given function f, the *graph* relation of f is the relation defined by

$$G(x_1,\ldots,x_n,y) \leftrightarrow f(x_1,\ldots,x_n) = y.$$

Let $f^*(x_1, \ldots, x_n, y) = f(x_1, \ldots, x_n)$. Then f^* is recursive if f is, since

$$f^* = \mathrm{Cn}[f, \mathrm{id}_1^{n+1}, \ldots, \mathrm{id}_n^{n+1}].$$

Now $f(x_1, \ldots, x_n) = y$ if and only if

$$f^*(x_1,\ldots,x_n,y) = \mathrm{id}_{n+1}^{n+1}(x_1,\ldots,x_n,y).$$

Indeed, the latter condition is essentially just a long-winded way of writing the former condition. But this shows that if f is a recursive total function, then the graph relation $f(x_1, \ldots, x_n) = y$ is obtainable from the identity relation $u = v$ by substituting the recursive total functions f^* and id_{n+1}^{n+1}. Thus *the graph relation of a recursive total function is a recursive relation*. More compactly, if less strictly accurately, we can summarize by saying that the graph relation $f(x) = y$ is obtained by substituting the recursive total function f in the identity relation. (This compact, slightly inaccurate manner of speaking, which will be used in future, suppresses mention of the role of the identity functions in the foregoing argument.)

Defining New Relations from Given Functions/Relations

- **The graph relation**
  - **For a given function *f*, the *graph relation* of *f* is the relation defined by $G(x_1, \ldots, x_n, y) \leftrightarrow f(x_1, \ldots, x_n) = y$.**
- **The negation or denial of a relation**
  - **Given a relation *R*, its *negation* or *denial* is the relation *S* that holds iff *R* does not: $S(x_1, \ldots, x_n) \leftrightarrow \sim(\neg)R(x_1, \ldots, x_n)$.**
- **Note**
  - **In accord with our official definition, relations are considered as sets of *k*-tuples, the negation is simply the complement.**

Defining New Relations from Given Functions/Relations

♣ **The conjunction of two relations**

- Given two relations R_1 and R_2, their *conjunction* is the relation S that holds IFF R_1 holds *and* R_2 holds:
  $S(x_1, ..., x_n) \leftrightarrow R_1(x_1, ..., x_n)$ &(∧) $R_2(x_1, ..., x_n)$.

♣ **The disjunction of two relations**

- Given two relations R_1 and R_2, their *disjunction* is the relation S that holds IFF R_1 holds *or* R_2 holds:
  $S(x_1, ..., x_n) \leftrightarrow R_1(x_1, ..., x_n) \vee R_2(x_1, ..., x_n)$.

♣ **Notes**

- Conjunction and disjunctions of more than two relations are similarly defined.
- Relations are considered as sets of k-tuples, the conjunction the intersection, and the disjunction the union.

Defining New Relations from Given Functions/Relations

♣ **The bounded universal/existential quantification of a relation**

- Given a relation $R(x_1, ..., x_n, u)$, by the relation obtained from R through *bounded universal quantification* we mean the relation S that holds of $x_1, ..., x_n, u$ iff for all $v < u$, the relation R holds of $x_1, ..., x_n, v$. We write or more fully:
  $S(x_1, ..., x_n, u) \leftrightarrow \forall v < u\ R(x_1, ..., x_n, v)$
  $S(x_1, ..., x_n, u) \leftrightarrow \forall v\ (v < u \rightarrow R(x_1, ..., x_n, v))$.
- By the relation obtained from R through *bounded existential quantification* we mean the relation S that holds of $x_1, ..., x_n, u$ iff for some $v < u$, the relation R holds of $x_1, ..., x_n, v$. We write or more fully:
  $S(x_1, ..., x_n, u) \leftrightarrow \exists v < u\ R(x_1, ..., x_n, v)$
  $S(x_1, ..., x_n, u) \leftrightarrow \exists v\ (v < u\ \&\ R(x_1, ..., x_n, v))$.
- The bounded quantifiers $\forall v \le u$ and $\exists v \le u$ are similarly defined.

Closure Properties of (Primitive) Recursive Relations [BBJ-ToC-07]

- **The following theorem and its corollary are stated for recursive relations (and recursive total functions), but hold equally for primitive recursive relations (and primitive recursive functions), by the same proofs, to include a bracketed '(primitive)' everywhere in the statement and proof of the result.**

7.4 Theorem (Closure properties of recursive relations).

(a) A relation obtained by substituting recursive total functions in a recursive relation is recursive.
(b) The graph relation of any recursive total function is recursive.
(c) If a relation is recursive, so is its negation.
(d) If two relations are recursive, then so is their conjunction.
(e) If two relations are recursive, then so is their disjunction.
(f) If a relation is recursive, then so is the relation obtained from it by bounded universal quantification.
(g) If a relation is recursive, then so is the relation obtained from it by bounded existential quantification.

Closure Properties of (Primitive) Recursive Relations [BBJ-ToC-07]

Proof:
(a), (b): These have already been proved.
(c): In the remaining items, we write simply x for $x_1, \ldots, x_n$. The characteristic function c^* of the negation or complement of R is obtainable from the characteristic function c of R by $c^*(x) = 1 \dot{-} c(x)$.
(d), (e): The characteristic function c^* of the conjunction or intersection of R_1 and R_2 is obtainable from the characteristic functions c_1 and c_2 of R_1 and R_2 by $c^*(x) = \min(c_1(x), c_2(x))$, and the characteristic function $c\dagger$ of the disjunction or union is similarly obtainable using max in place of min.
(f), (g): From the characteristic function $c(x, y)$ of the relation $R(x, y)$ the characteristic functions u and e of the relations $\forall v \le y\ R(x_1, \ldots, x_n, v)$ and $\exists v \le y\ R(x_1, \ldots, x_n, v)$ are obtainable as follows:

$$u(x, y) = \prod_{i=0}^{y} c(x, i) \qquad e(x, y) = \mathrm{sg}\left(\sum_{i=0}^{y} c(x, i)\right)$$

where the summation ($\sum$) and product ($\prod$) notation is as in Proposition 6.5. For the product will be 0 if any factor is 0, and will be 1 if and only if all factors are 1; while the sum will be positive if any summand is positive. For the strict bounds $\forall v < y$ and $\exists v < y$ we need only replace y by $y \dot{-} 1$.

Closure Properties of (Primitive) Recursive Functions [BBJ-ToC-07]

7.5 Example (Primality). Recall that a natural number x is prime if $x > 1$ and there do not exist any u, v both $< x$ such that $x = u \cdot v$. The set P of primes is primitive recursive, since we have

$$P(x) \leftrightarrow 1 < x\ \&\ \forall u < x\ \forall v < x(u \cdot v \neq x).$$

Here the relation $1 < x$ is the result of substituting const_1 and id into the relation $y < x$, which we know to be primitive recursive from Example 7.1, and so this relation is primitive recursive by clause (a) of the theorem. The relation $u \cdot v = x$ is the graph of a primitive recursive function, namely, the product function; hence this relation is primitive recursive by clause (b) of the theorem. So P is obtained by negation, bounded universal quantification, and conjunction from primitive recursive relations, and is primitive recursive by clauses (c), (d), and (f) of the theorem.

Closure Properties of (Primitive) Recursive Functions [BBJ-ToC-07]

例 7.5（素数性）回忆素数的定义，如果 $x > 1$ 并且不存在小于 x 的正整数 u, v 使得 $u \cdot v = x$，则 x 是素数。素数集合 P 是原始递归的，因为

$$P(x) \leftrightarrow 1 < x\ \&\ \forall u < x\ \forall v < x(u \cdot v \neq x)$$

这里关系 $1 < x$ 是 const_1 和 id 代入关系 $x < y$ 的结果，由例 7.1 可知关系 $x < y$ 是原始递归的[*]，因此由上面定理中的(a)得出关系 $1 < x$ 是原始递归的。关系 $u \cdot v = x$ 是乘法函数的图，而乘法函数是原始递归的，由上面定理中的(b)得出关系 $u \cdot v = x$ 是原始递归的。从原始递归关系出发，通过否定、有界全称量化、合取得到了 P，因此，由上面定理中的(c), (d), (f)得出 P 是原始递归的。

Closure Properties of (Primitive) Recursive Functions [BBJ-ToC-07]

7.6 Corollary (Bounded minimization and maximization). Given a (primitive) recursive relation R, let

$$\mathrm{Min}[R](x_1,\ldots,x_n,w)=\begin{cases}\text{the smallest } y\le w \text{ for which} & \\ \quad R(x_1,\ldots,x_n,y) & \text{if such a } y \text{ exists}\\ w+1 & \text{otherwise}\end{cases}$$

and

$$\mathrm{Max}[R](x_1,\ldots,x_n,w)=\begin{cases}\text{the largest } y\le w \text{ for which} & \\ \quad R(x_1,\ldots,x_n,y) & \text{if such a } y \text{ exists}\\ 0 & \text{otherwise.}\end{cases}$$

Then Min[R] and Max[R] are (primitive) recursive total functions.

Closure Properties of (Primitive) Recursive Functions [BBJ-ToC-07]

Proof: We give the proof for Min. Write x for $x_1,\ldots,x_n$. Consider the (primitive) recursive relation $\forall t\le y\ {\sim}R(x,t)$, and let c be its characteristic function. If there is a smallest $y\le w$ such that $R(x,y)$, then abbreviating $c(x,i)$ to $c(i)$ we have

$$c(0)=c(1)=\cdots=c(y-1)=1 \qquad c(y)=c(y+1)=\cdots=c(w)=0.$$

So c takes the value 1 for the y numbers $i<y$, and the value 0 thereafter. If there is no such y, then

$$c(0)=c(1)=\cdots=c(w)=1.$$

So c takes the value 1 for all $w+1$ numbers $i\le w$. In either case

$$\mathrm{Min}[R](x,w)=\sum_{i=0}^{w}c(x,i)$$

and is therefore (primitive) recursive. The proof for Max is similar, and is left to the reader.

Closure Properties of (Primitive) Recursive Functions [BBJ-ToC-07]

7.7 Example (Quotients and remainders). Given natural numbers x and y with $y>0$, there are unique natural numbers q and r such that $x=q\cdot y+r$ and $r<y$. They are called the *quotient* and *remainder* on division of x by y. Let quo(x,y) be the quotient on dividing x by y if $y>0$, and set quo$(x,0)=0$ by convention. Let rem(x,y) be the remainder on dividing x by y if $y>0$, and set rem$(x,0)=x$ by convention. Then quo is primitive recursive, as an application of bounded maximization, since $q\le x$ and q is the largest number such that $q\cdot y\le x$.

$$\mathrm{quo}(x,y)=\begin{cases}\text{the largest } z\le x \text{ such that } y\cdot z\le x & \text{if } \ y\neq 0\\ 0 & \text{otherwise.}\end{cases}$$

We apply the preceding corollary (in its version for primitive recursive functions and relations). If we let $Rxyz$ be the relation $y\cdot z\le x$, then quo$(x,y)=\mathrm{Max}[R](x,y,x)$, and therefore quo is primitive recursive. Also rem is primitive recursive, since rem$(x,y)=x\dot{-}(\mathrm{quo}(x,y)\cdot y)$. Another notation for rem(x,y) is x mod y.

Closure Properties of (Primitive) Recursive Functions [BBJ-ToC-07]

例 7.7 （商和余数）给定自然数 x 和 y，其中 $y>0$，存在唯一的自然数 q 和 r 使得 $x=q\cdot y+r$ 且 $r<y$。它们分别被称为用 y 除 x 的商和余数。当 $y>0$ 时令 quo(x,y)是用 y 除 x 的商，并约定 quo$(x,0)=0$。当 $y>0$ 时令 rem(x,y)是用 y 除 x 的余数，并约定 rem$(x,0)=x$，则 quo 是原始递归的，因为 $q\leqslant x$ 且 q 是使得 $q\cdot y\leqslant x$ 的最大数，所以应用有界极大化可得到 quo。

$$\mathrm{quo}(x,y)=\begin{cases}\text{使得} y\cdot z\leqslant x\text{且}z\leqslant x\text{的最大}z & \text{如果} y\neq 0\\ 0 & \text{否则}\end{cases}$$

这里应用了前面的推论（关于原始递归函数和原始递归关系的版本）。如果设 $Rxyz$ 是关系 $y\cdot z\leqslant x$，则 quo$(x,y)=\mathrm{Max}[R](x,y,x)$，因此 quo 是原始递归的。因为 rem$(x,y)=x\dot{-}(\mathrm{quo}(x,y)\cdot y)$，所以 rem 也是原始递归的。rem$(x,y)$的另一种记法是 x mod y。

Closure Properties of (Primitive) Recursive Functions [BBJ-ToC-07]

7.8 Corollary. Suppose that f is a regular primitive function and that there is a primitive recursive function g such that the least y with $f(x_1,\ldots,x_n,y)=0$ is always less than $g(x_1,\ldots,x_n)$. Then Mn[f] is not merely recursive but primitive recursive.

Proof: Let $R(x_1,\ldots,x_n,y)$ hold if and only if $f(x_1,\ldots,x_n,y)=0$. Then

$$\mathrm{Mn}[f](x_1,\ldots,x_n)=\mathrm{Min}[R](x_1,\ldots,x_n,g(x_1,\ldots,x_n)).$$

7.9 Proposition. Let R be an $(n+1)$-place recursive relation. Define a total or partial function r by

$$r(x_1,\ldots,x_n)=\text{the least } y \text{ such that } R(x_1,\ldots,x_n,y).$$

Then r is recursive.

Proof: The function r is just Mn[c], where c is the characteristic function of ${\sim}R$.

Closure Properties of (Primitive) Recursive Functions [BBJ-ToC-07]

7.10 Example (The next prime). Let $f(x)=$ the least y such that $x<y$ and y is prime. The relation

$$x<y \ \&\ y \text{ is prime}$$

is primitive recursive, using Example 7.5. Hence the function f is recursive by the preceding proposition. There is a theorem in Euclid's *Elements* that tells us that for any given number x there exists a prime $y>x$, from which we know that our function f is total. But actually, the proof in Euclid shows that there is a prime $y>x$ with $y\le x!+1$. Since the factorial function is primitive recursive, the Corollary 7.8 applies to show that f is actually *primitive* recursive.

例 7.10 （下一个素数）设 $f(x)=$ 使得 $x<y$ 且 y 是素数的最小 y。根据例 7.5，关系

$$x<y\ \&\ y\text{是素数}$$

是原始递归的。因此，根据前一命题，f 是递归函数。欧几里得《几何原本》中的定理指出，对于任意给定数 x，存在素数 $y>x$，由此得出，f 是全函数。实际上，欧几里得的证明中指出，存在素数 $y>x$ 满足 $y\leqslant x!+1$。因为阶乘函数是原始递归的，所以利用推论 7.8 可证明，实际上 f 是原始递归的。

Closure Properties of (Primitive) Recursive Functions [BBJ-ToC-07]

7.11 Example (Logarithms). Subtraction, the inverse operation to addition, can take us beyond the natural numbers to negative integers; but we have seen there is a reasonable modified version $\dot{-}$ that stays within the natural numbers, and that it is primitive recursive. Division, the inverse operation to multiplication, can take us beyond the integers to fractional rational numbers; but again we have seen there is a reasonable modified version quo that is primitive recursive. Because the power or exponential function is not commutative, that is, because in general $x^y \neq y^x$, there are *two* inverse operations: the yth root of x is the z such that $z^y = x$, while the base-x logarithm of y is the z such that $x^z = y$. Both can take us beyond the rational numbers to irrational real numbers or even imaginary and complex numbers. But again there is a reasonable modified version, or several reasonable modified versions. Here is one for the logarithms

$$\mathrm{lo}(x, y) = \begin{cases} \text{the greatest } z \leq x \text{ such that } y^z \text{ divides } x & \text{if } x, y > 1 \\ 0 & \text{otherwise} \end{cases}$$

where 'divides x' means 'divides x without remainder'. Clearly if $x, y > 1$ and y^z divides x, z must be (quite a bit) less than x. So we can agrue as in the proof of 7.7 to show that lo is a primitive recursive function. Here is another reasonable modified logarithm function:

$$\mathrm{lg}(x, y) = \begin{cases} \text{the greatest } z \text{ such that } y^z \leq x & \text{if } x, y > 1 \\ 0 & \text{otherwise.} \end{cases}$$

The proof that lg is primitive recursive is left to the reader.

Closure Properties of (Primitive) Recursive Functions [BBJ-ToC-07]

例 7.11 （对数）加法的逆运算减法的运算结果能超出自然数得到负整数，但是，我们已经看到，适当修改减法得到的原始递归函数 $\dot{-}$ 能使得运算结果仍然保持为自然数。乘法的逆运算除法的运算结果能超出整数得到有理分数，而我们又一次看到，适当修改除法能得到原始递归函数 quo。因为幂函数不是可交换的，即一般说来，$x^y \neq y^x$，因此它有两个逆运算：使得 $z^y = x$ 的 x 的 y 次根 z 和使得 $x^z = y$ 的以 x 为底的 y 的对数 z。两者的运算结果都能超出有理数得到无理数甚至虚数。可以有几种合理的修改方案。对数函数的一个适当修改是：

$$\mathrm{lo}(x, y) = \begin{cases} \text{使得} y^z \text{能整除} x \text{的最大} z & \text{如果} x, y > 1 \\ 0 & \text{否则} \end{cases}$$

其中“整除 x”是指“除 x 没有余数”。显然，如果 $x, y > 1$ 且 y^z 能整除 x，z 必须小于 x，哪怕只小一点。因此根据推论 7.8，lo 是原始递归函数。下面是对数函数的另一个适当修改：

$$\mathrm{lg}(x, y) = \begin{cases} \text{使得} y^z \leq x \text{的最大} z & \text{如果} x, y > 1 \\ 0 & \text{否则} \end{cases}$$

证明 lg 是原始递归函数的任务留给读者。

Coding Finite Sequences of Natural Numbers by Single Natural Numbers [BBJ-ToC-07]

♣Coding finite sequences of natural numbers by single natural numbers

- **The coding we adopt is based on the fact that each positive integer can be written in one and only one way as a product of powers of larger and larger primes. Let $\pi(n)$ be the nth prime (counting 2 as the 0th). Specifically:**

$$(a_0, a_1, \ldots, a_{n-1}) \text{ is coded by } 2^n 3^{a_0} 5^{a_1} \cdots \pi(n)^{a_{n-1}}$$

7.12 Example (The nth prime). Let $\pi(n)$ be the nth prime, counting 2 as the 0th, so $\pi(0) = 2$, $\pi(1) = 3$, $\pi(2) = 5$, $\pi(3) = 7$, and so on. This function is primitive recursive.

7.13 Example (Length). There is a primitive recursive function lh such that if s codes a sequence $(a_0, a_1, \ldots, a_{n-1})$, then the value $\mathrm{lh}(s)$ is the *length* of that sequence.

7.14 Example (Entries). There is a primitive recursive function ent such that if s codes a sequence $(a_0, a_1, \ldots, a_{n-1})$, then for each $i < n$ the value of $\mathrm{ent}(s, i)$ is the ith *entry* in that sequence (counting a_0 as the 0th).

Coding Finite Sequences of Natural Numbers by Single Natural Numbers [BBJ-ToC-07]

♣Coding finite sequences of natural numbers by single natural numbers

Proofs

Example 7.12. $\pi(0) = 2, \pi(x') = f(\pi(x))$, where f is the next prime function of Example 7.10. The form of the definition is similar to that of the factorial function: see Example 6.4 for how to reduce definitions of this form to the official format for recursion.

Example 7.13. $\mathrm{lh}(s) = \mathrm{lo}(s, 2)$ will do, where lo is as in Example 7.11. Applied to

$$2^n 3^{a_0} 5^{a_1} \cdots \pi(n)^{a_{n-1}}$$

this function yields n.

Example 7.14. $\mathrm{ent}(s, i) = \mathrm{lo}(s, \pi(i + 1))$ will do. Applied to

$$2^n 3^{a_0} 5^{a_1} \cdots \pi(n)^{a_{n-1}}$$

and i, this function yields a_i.

Coding Finite Sequences of Natural Numbers by Single Natural Numbers [BBJ-ToC-07]

♣Coding finite sequences of natural numbers by single natural numbers

例 7.12 （第 n 个素数）设 $\pi(n)$ 是第 n 个素数，将 2 看做第 0 个素数，因此 $\pi(0) = 2$，$\pi(1) = 3$，$\pi(2) = 5$，$\pi(3) = 7$，等等。这个函数是原始递归的。

例 7.13 （长度）存在原始递归函数 lh 使得，如果序列 $(a_0, a_1, \cdots, a_{n-1})$ 的编码为 s，则函数值 $\mathrm{lh}(s)$ 是该序列的长度。

例 7.14 （项）存在原始递归函数 ent 使得，如果序列 $(a_0, a_1, \cdots, a_{n-1})$ 的编码为 s，则对于每个 $i < n$，函数值 $\mathrm{ent}(s, i)$ 是该序列的第 i 项（将 a_0 看做第 0 项）。

证明

例 7.12 $\pi(0) = 2$，$\pi(x') = f(\pi(x))$，其中 f 是例 7.10 中的求下一个素数的函数。这个定义的形式类似于阶乘函数定义的形式：怎样将这种形式的定义归约为递归的正式格式，见例 6.4。

例 7.13 令 $\mathrm{lh}(s) = \mathrm{lo}(s, 2)$ 即可，其中 lo 如例 7.11 所定义。函数 lh 在自变量

$$2^n 3^{a_0} 5^{a_1} \cdots \pi(n)^{a_{n-1}}$$

处求值得到 n。

例 7.14 令 $\mathrm{ent}(s, i) = \mathrm{lo}(s, \pi(i+1))$ 即可，则 $\mathrm{ent}(2^n 3^{a_0} 5^{a_1} \cdots \pi(n)^{a_{n-1}}, i) = a_i$。

Semi-Recursive Sets

♣Semi-recursive sets

- **A set of natural numbers is *(positively) effectively semi-decidable*, or simply *semi-recursive*, if there is an effective procedure that, applied to any natural number, will if the number is in the set in a finite amount of time (steps) give the answer 'yes', but will if the natural number is not in the set never give an answer.**

♣Examples

- **For instance, the domain of an effectively computable partial function *f* is always effectively semi-decidable: the procedure for determining whether *n* is in the domain of *f* is simply to try to compute the value *f*(*n*); if and when we succeed, we know that *n* is in the domain; but if *n* is not in the domain, we never succeed.**

Semi-Recursive Relations

♣ **Effectively decidable relations from semi-decidable sets**

- **If S is a semi-decidable set then we have $S(x) \leftrightarrow \exists t\, R(x, t)$ where R is the *effectively decidable* relation 'by t steps of computation we obtain the answer "yes"'.**

♣ **Semi-recursive sets from effectively decidable relations**

- **Conversely, if R is an effectively decidable relation of any kind, and S is the relation obtained from R by (unbounded) existential quantification, then S is effectively semi-decidable: we can attempt to determine whether n is in S by checking whether $R(n, 0)$ holds, and if not, whether $R(n, 1)$ holds, and if not, whether $R(n, 2)$ holds, and so on. If n is in S, we must eventually find a t such that $R(n, t)$, and will thus obtain the answer 'yes'; but if n is not in S, we go on forever without obtaining an answer.**

Semi-Recursive Relations

♣ **Semi-recursive relations**

- **Thus we may characterize the effectively semi-decidable sets as those obtained from two-ary effectively decidable relations by existential quantification, and more generally, the n-ary effectively semi-decidable relations as those obtained from $(n+1)$-ary effectively decidable relations by existential quantification.**
- **An n-ary relation S on natural numbers is *(positively) recursively semi-decidable*, or simply *semi-recursive*, if it is obtainable from an $(n+1)$-ary recursive relation R by existential quantification: $S(x_1, ..., x_n) \leftrightarrow \exists y\, R(x_1, ..., x_n, y)$.**
- **A y such that R holds of the x_i and y may be called a '*witness*' to the relation S holding of the x_i (provided we understand that when the witness is a number rather than a person, a witness only testifies to what is true).**

Closure Properties of Semi-recursive Relations [BBJ-ToC-07]

- **The closure properties of recursive relations established in Theorem 7.4 can be used to establish a similar but not identical list of properties of semi-recursive relations.**

7.15 Corollary (Closure properties of semirecursive relations).

(a) Any recursive relation is semirecursive.
(b) A relation obtained by substituting recursive total functions in a semirecursive relation is semirecursive.
(c) If two relations are semirecursive, then so is their conjunction.
(d) If two relations are semirecursive, then so is their disjunction.
(e) If a relation is semirecursive, then so is the relation obtained from it by bounded universal quantification.
(f) If a relation is semirecursive, then so is the relation obtained from it by existential quantification.

Closure Properties of Semi-recursive Relations [BBJ-ToC-07]

Proof: We write simply x for $x_1, \ldots, x_n$.

(a): If Rx is a recursive relation, then the relation S given by $Sxy \leftrightarrow (Rx \,\&\, y = y)$ is also recursive, and we have $R(x) \leftrightarrow \exists y\, Sxy$.

(b): If Rx is a semirecursive relation, say $Rx \leftrightarrow \exists y Sxy$ where S is recursive, and if $R^*x \leftrightarrow Rf(x)$, where f is a recursive total function, then the relation S^* given by $S^*xy \leftrightarrow Sf(x)y$ is also recursive, and we have $R^*x \leftrightarrow \exists y\, S^*xy$ and R^* is semi-recursive.

(c): If R_1x and R_2x are semirecursive relations, say $R_ix \leftrightarrow \exists y\, S_ixy$ where S_1 and S_2 are recursive, then the relation S given by $Sxw \leftrightarrow \exists y_1 < w\ \exists y_2 < w(S_1xy_1 \,\&\, S_2xy_2)$ is also recursive, and we have $(R_1x \,\&\, R_2y) \leftrightarrow \exists w\ Sxw$. We are using here the fact that for any two numbers y_1 and y_2, there is a number w greater than both of them.

(d): If R_i and S_i are as in (c), then the relation S given by $Sxy \leftrightarrow (S_1xy \vee S_2xy)$ is also recursive, and we have $(R_1y \vee R_2x) \leftrightarrow \exists y\ Sxy$.

(e): If Rx is a semirecursive relation, say $Rx \leftrightarrow \exists y\ Sxy$ where S is recursive, and if $R^*x \leftrightarrow \forall u < x Ru$, then the relation S^* given by $S^*xw \leftrightarrow \forall u < x\ \exists y < w\ Suy$ is also recursive, and we have $R^*x \leftrightarrow \exists w\ S^*xw$. We are using here the fact that for any finite number of numbers $y_0, y_1, \ldots, y_x$ there is a number w greater than all of them.

(f): If Rxy is a semirecursive relation, say $Rxy \leftrightarrow \exists z\, Sxyz$ where S is recursive, and if $R^*x \leftrightarrow \exists y\ Rxy$, then the relation S^* given by $S^*xw \leftrightarrow \exists y < w\ \exists z < w\ Sxyz$ is also recursive, and we have $R^*x \leftrightarrow \exists w\ S^*xw$.

Yielding New Recursive Relations by Semi-Recursive Relations

♣ **Yielding new recursive relations by semi-recursive relations**

- **Intuitively, if we have a procedure that will eventually tell us when a number is in a set (but will tell us nothing if it is not), and *also* have a procedure that will eventually tell us when a number is not in a set (but will tell us nothing if it is), then by combining them we can get a procedure that will tell us *whether or not* a number is in the set: apply both given procedures, and eventually one or the other must give us an answer.**
- **In jargon, if a set and its complement are both effectively semi-decidable, then the set is decidable.**

Yielding New Recursive Relations by Semi-Recursive Relations [BBJ-ToC-07]

7.16 Proposition (Complementation principle, or Kleene's theorem). If a set and its complement are both semirecursive, then the set (and hence also its complement) is recursive.

Proof: If Rx and $\sim Rx$ are both semirecursive, say $Rx \leftrightarrow \exists y\ S^+xy$ and $\sim Rx \leftrightarrow \exists y\ S^-xy$, then the relation S^* given by $S^*xy \leftrightarrow (S^+xy \vee S^-xy)$ is recursive, and if f is the function defined by letting $f(x)$ be the least y such that S^*xy, then f is a recursive total function. But then we have $Rx \leftrightarrow S^+xf(x)$, showing that R is obtainable by substituting a recursive total function in a recursive relation, and is therefore recursive.

Yielding New Recursive Relations by Semi-Recursive Relations [BBJ-ToC-07]

7.17 Proposition (First graph principle). If the graph relation of a total or partial function f is semirecursive, then f is a recursive total or partial function.

Proof: Suppose $f(x) = y \leftrightarrow \exists z\, Sxyz$, where S is recursive. We first introduce two auxiliary functions:

$$g(x) = \begin{cases} \text{the least } w \text{ such that} \\ \quad \exists y < w\ \exists z < w\ Sxyz & \text{if such a } w \text{ exists} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$h(x, w) = \begin{cases} \text{the least } y < w \text{ such that} \\ \quad \exists z < w\ Sxyz & \text{if such a } y \text{ exists} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Here the relations involved are *recursive*, and not just semirecursive, since they are obtained from S by *bounded*, not unbounded, existential quantification. So g and h are recursive. And a little thought shows that $f(x) = h(x, g(x))$, so f is recursive also.

Further Examples [BBJ-ToC-07]

7.18 Example (First and last). There are primitive recursive functions fst and lst such that if s codes a sequence $(a_0, a_1, \ldots, a_{n-1})$, then fst($s$) and lst($s$) are the first and last entries in that sequence.

7.19 Example (Extension). There is a primitive recursive function ext such that if s codes a sequence $(a_0, a_1, \ldots, a_{n-1})$, then for any b, ext(s, b) codes the *extended* sequence $(a_0, a_1, \ldots, a_{n-1}, b)$.

7.20 Example (Concatenation). There is a primitive recursive function conc such that if s codes a sequence $(a_0, a_1, \ldots, a_{n-1})$ and t codes a sequence $(b_0, b_1, \ldots, b_{m-1})$, then conc ($s$, t) codes the *concatenation* $(a_0, a_1, \ldots, a_{n-1}, b_0, b_1, \ldots, b_{m-1})$ of the two sequences.

Further Examples [BBJ-ToC-07]

Proofs

Example 7.18. $\text{fst}(s) = \text{ent}(s, 0)$ and $\text{lst}(s) = \text{ent}(s, \text{lh}(s) \dot{-} 1)$ will do.

Example 7.19. $\text{ext}(s, b) = 2 \cdot s \cdot \pi(\text{lh}(s) + 1)^b$ will do. Applied to

$$2^n 3^{a_0} 5^{a_1} \cdots \pi(n)^{a_{n-1}}$$

this function yields

$$2^{n+1} 3^{a_0} 5^{a_1} \cdots \pi(n)^{a_{n-1}} \pi(n+1)^b.$$

Example 7.20. A head-on approach here does not work, and we must proceed a little indirectly, first introducing an auxiliary function such that

$$g(s, t, i) = \text{ the code for } (a_0, a_1, \ldots, a_{n-1}, b_0, b_1, \ldots, b_{i-1}).$$

We can then obtain the function we really want as $\text{conc}(s, t) = g(s, t, \text{lh}(t))$. The auxiliary g is obtained by recursion as follows:

$$g(s, t, 0) = s$$
$$g(s, t, i') = \text{ext}(g(s, t, i), \text{ent}(t, i)).$$

Two more we leave entirely to the reader.

Further Examples [BBJ-ToC-07]

例 7.18（第一项和最后一项）存在原始递归函数 fst 和 lst 使得，如果 s 是序列（$a_0, a_1, \cdots, a_{n-1}$）的编码，则 fst($s$)和 lst($s$)分别是该序列的第一项和最后一项。

例 7.19 （扩展）存在原始递归函数 ext 使得，如果 s 是序列（$a_0, a_1, \cdots, a_{n-1}$）的编码，则对于任意 b，ext(s, b)是扩展序列（$a_0, a_1, \cdots, a_{n-1}, b$）的编码。

例 7.20 （连接）存在原始递归函数 conc 使得，如果 s 是序列（$a_0, a_1, \cdots, a_{n-1}$）的编码，$t$ 是序列（$b_0, b_1, \cdots, b_{m-1}$）的编码，则 conc($s$, t)是这两个序列的连接（$a_0, a_1, \cdots, a_{n-1}, b_0, b_1, \cdots, b_{m-1}$）的编码。

证明

例 7.18 $\text{fst}(s) = \text{ent}(s, 0)$且 $\text{lst}(s) = \text{ent}(s, \text{lh}(s) \dot{-} 1)$。

例 7.19 $\text{ext}(s, b) = 2 \cdot s \cdot \pi(\text{lh}(s)+1)^b$。这个函数在自变量

$$2^n 3^{a_0} 5^{a_1} \cdots \pi(n)^{a_{n-1}}$$

处的函数值是：

$$2^{n+1} 3^{a_0} 5^{a_1} \cdots \pi(n)^{a_{n-1}} \pi(n+1)^b$$

例 7.20 上例所用的直接方法已不适用，必须采取间接的方法，引进一个辅助函数：

$g(s, t, i) =$ 序列（$a_0, a_1, \cdots, a_{n-1}, b_0, b_1, \cdots, b_{i-1}$）的编码

然后令 $\text{conc}(s, t) = g(s, t, \text{lh}(t))$得到结果。辅助函数 g 递归定义如下：

$$g(s, t, 0) = s$$
$$g(s, t, i') = \text{ext}(g(s, t, i), \text{ent}(t, i))$$

下面两个例子的证明留给读者。

Further Examples [BBJ-ToC-07]

7.21 Example (Truncation). There is a primitive recursive function tr such that if s codes a sequence $(a_0, a_1, \ldots, a_{n-1})$ and $m \leq n$, then tr(s, m) codes the *truncated* sequence $(a_0, a_1, \ldots, a_{m-1})$.

7.22 Example (Substitution). There is a primitive recursive function sub such that if s codes a sequence $(a_1, \ldots, a_k)$, and c and d are any natural numbers, then sub(s, c, d) codes the sequence that results upon taking s and substituting for any entry that is equal to c the number d instead.

例 7.21 （截短）存在原始递归函数 tr 使得，如果 s 是序列$(a_0, a_1, \cdots, a_{n-1})$的编码且 $m \leq n$，则 tr(s, m)是被截短的序列$(a_0, a_1, \cdots, a_{m-1})$的编码。

例 7.22 （代换）存在原始递归函数 sub 使得，如果 s 是序列$(a_1, \cdots, a_k)$的编码，c 和 d 是任意自然数，则 sub(s, c, d)是将该序列中的所有 c 代换为 d 得出的序列的编码。

Further Examples [BBJ-ToC-07]

7.24 Proposition. It is impossible to obtain the sum or addition function from the basic functions (zero, successor, and identity) by composition, without using recursion.

Proof: To prove this negative result we claim something positive, that if f belongs to the class of functions that can be obtained from the basic functions using only composition, then there is a positive integer a such that for all $x_1, \ldots, x_n$ we have $f(x_1, \ldots, x_n) < x + a$, where x is the largest of $x_1, \ldots, x_n$. No such a can exist for the addition function, since $(a+1) + (a+1) > (a+1) + a$, so it will follow that the addition function is not in the class in question—provided we can prove our claim. The claim is certainly true for the zero function (with $a = 1$), and for the successor function (with $a = 2$), and for each identity function (with $a = 1$ again). Since every function in the class we are interested in is built up step by step from these functions using composition, it will be enough to show if the claim holds for given functions, it holds for the function obtained from them by composition.

Further Examples [BBJ-ToC-07]

So consider a composition

$$h(x_1, \ldots, x_n) = f(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n)).$$

Suppose we know

$$g_i(x_1, \ldots, x_n) < x + a_j \qquad \text{where } x \text{ is the largest of the } x_j$$

and suppose we know

$$f(y_1, \ldots, y_m) < y + b \qquad \text{where } y \text{ is the largest of the } y_i.$$

We want to show there is a c such that

$$h(x_1, \ldots, x_n) < x + c \qquad \text{where } x \text{ is the largest of the } x_j.$$

Let a be the largest of $a_1, \ldots, a_m$. Then where x is the largest of the x_j, we have

$$g_i(x_1, \ldots, x_n) < x + a$$

so if $y_i = g_i(x_1, \ldots, x_n)$, then where y is the largest of the y_i, we have $y < x + a$. And so

$$h(x_1, \ldots, x_n) = f(y_1, \ldots, y_m) < (x + a) + b = x + (a + b)$$

and we may take $c = a + b$.

An Introduction to the Theory of Computation

- Enumerability and Diagonalization
- Finite Automata and Regular Languages
- Pushdown Automata and Context-Free Languages
- Computation: Turing Machines
- Computation: Turing-Computability (Turing-Decidability)
- Computation: Reducibility and Turing-Reducibility
- Computation: Recursive Functions
- Computation: Recursive Sets and Relations
- Equivalent Definitions of Computability
- Advanced Topics in Computability Theory
- Computational Complexity
- Time Complexity
- Space Complexity
- Intractability
- Advanced Topics in Complexity Theory