CDM
Primitive Recursion

Klaus Sutner
Carnegie Mellon University
Spring 2021

Computability, Definition

We need a formal definition of computability that

- is easy to understand and apply, and
- matches our intuitive notion of computability.

There are many plausible approaches, we'll start with a model for arithmetic functions that dates back to the 19th century and is quite straightforward. Additionally, it is more palatable that machine models to those used to program.

Knee-Jerk Reaction

Computable means: can be done, in principle, by a standard digital computer.

This sounds good to anyone who has ever written and executed a program; sadly, there are lots of problems with this approach.

First, the hedge "in principle" means you really have to abstract away from a concrete physical device (time, space, mass, energy, ...).

Then there is the question which operating system, which programming language, which compiler? These typically have no clear semantics, so what exactly are we defining?

But Why?

Can't we get away with an informal, wishy-washy definition?

Yes and No, but mostly: No.

Informal is typically good enough for positive results: such-and-such thing is computable.

But for negative results we need real foundations: such-and-such problem fails to be computable, or fails to be computable given particular resources (complexity theory). As we now understand, the latter type of question can be breathtakingly difficult.

Math versus CS

Aka computability versus algorithms.

Implementation details are usually of little interest in mathematics, it only matters whether a function is computable or not. Computability is a central foundational issue, but does not require detailed analysis.

CS is a bit different here, computability alone is typically of interest only as a very first step (e.g., when one establishes the decidability of some problem), to be followed by a careful effort to streamline the computations (so as to keep resource bounds low).

This second step leads into the realm of algorithms, which should not be confused with computable functions. Arguably, the concept of algorithm is much more complicated and currently only ill-defined notion. We'll focus on computability.

Computable Arithmetic Functions

For the time being we consider only one data type: the natural numbers $\mathbb{N}$. The corresponding functions are called arithmetic functions or number theoretic functions: Some examples are familiar to any kindergartener: addition, multiplication, squaring, roots, exponentiation and so on.

$$f : \mathbb{N}^n \to \mathbb{N}$$

We introduce a model of computation that is designed to work particularly well with these, no input/output coding is required.

For the time being, all our functions will be total.

Primitive Recursion

The main idea behind our first model is quite straightforward for anyone who has every used a modern programming language: we will define a function $f : \mathbb{N} \times \mathbb{N}^n \to \mathbb{N}$ by

- defining $f(0, \boldsymbol{y})$ explicitly, and
- defining $f(x+1, \boldsymbol{y})$ in terms of $f(x, \boldsymbol{y})$.

This should produce computable functions: we can either compute $f(n, \boldsymbol{y})$, $f(n-1, \boldsymbol{y})$, $f(n-2, \boldsymbol{y})$ ... top-down (slightly complicated, requires a recursion stack), or we can compute bottom-up $f(0, \boldsymbol{y})$, $f(1, \boldsymbol{y})$, $f(2, \boldsymbol{y})$ ... This requires no more than a loop.

Later we will see more complicated forms of recursion.

Details: Primitive Recursive Functions

Interestingly, Gödel encountered the problem of defining computable functions working on his seminal incompleteness theorem. He introduced a class of "very simple," easily describable functions, that are now called primitive recursive functions.

Recursion is the key idea, but we need a few more ingredients such as composition and projections.

It will always be crystal clear that our functions are intuitively computable.

Composition

Given functions $g_i : \mathbb{N}^m \to \mathbb{N}$ for $i = 1, \ldots, n$, $h : \mathbb{N}^n \to \mathbb{N}$, we define a new function $f : \mathbb{N}^m \to \mathbb{N}$ by composition as follows:

$$f(\boldsymbol{x}) = h(g_1(\boldsymbol{x}), \ldots, g_n(\boldsymbol{x}))$$

Notation: we write $\mathrm{Comp}[h, g_1, \ldots, g_n]$ or simply $h \circ (g_1, \ldots, g_n)$ inspired by the the well-known special case $m = 1$:

$$(h \circ g)(x) = h(g(x)).$$

It is clear that computability is closed wrto composition: output can be re-used as input.

Digression: Arity

When dealing with functions or relations, it is sometimes convenient to be able to express the arity as part of the notation used.

We will use a superscript (n) for this purpose:

$f^{(n)}$ a function of arity n

In particular write $c_a^{(n)}$ for the n-ary constant map $\boldsymbol{x} \mapsto a$.

We will call $c_a^{(0)}$ a hard constant.

Bureaucracy: Projections

Unfortunately, composition by itself is not quite enough.

Suppose we have a binary version add of addition, and want to define a ternary version. No problem:

$$\mathrm{add}^{(3)}(x, y, z) = \mathrm{add}^{(2)}(x, \mathrm{add}^{(2)}(y, z))$$

But, this is **not** allowed according to our definition of composition; just try.

We need a simple auxiliary tool, so-called projections:

$$P_i^n : \mathbb{N}^n \to \mathbb{N} \qquad P_i^n(x_1, \ldots, x_n) = x_i$$

where $1 \leq i \leq n$ for the projections.

Describing Functions

Now we can write

$$\mathsf{add}^{(3)} = \mathsf{add}^{(2)} \circ (P_1^3, \mathsf{add}^{(2)} \circ (P_2^3, P_3^3))$$

Note that no variables are needed in this notation system.

In general, we will prefer the informal notation, but you should know how to use projections to write formally correct terms.

Clones

A clone or function algebra is a collection of arithmetic functions that contains all projections and is closed under composition.

More generally, for any set A, define the collection of all finitary functions over A as

$$\mathfrak{F}_A = \bigcup_{n \geq 0} (A^n \to A)$$

Definition

A clone (over A) is a subset $\mathcal{C} \subseteq \mathfrak{F}_A$ that contains all projections and is closed under composition.

For example, all projections form a clone, as do all arithmetic functions.

Nullary Functions

Note that we allow hard constants, null-ary functions in $A^0 \to A$.

We will write $f()$ or $f(*)$ when we evaluate such functions.

In the literature, you will also find clones without null-ary functions

$$\mathcal{C} \subseteq \mathfrak{F}_A^{(+)} = \bigcup_{n > 0} (A^n \to A)$$

This is mostly a technical detail, but one should be aware of the issue.

Actually, this is exactly the kind of pesky detail that makes programming quite so difficult.

Nullary Composition

Recall composition: $f^{(n)}$, $g_i^{(m)}$, $i \in [n]$, produces $f \circ (g_1, \ldots, g_n) \in \mathfrak{F}_A^{(m)}$ where $n, m \geq 0$.

It is worthwhile to consider the special case where f or the g_i are nullary.

Case: $n = 0$

Then for $m \geq 1$ we have $c_{f(*)}^{(m)} \in \mathcal{C}$.

Case: $m = 0$

Then for $n \geq 1$ we have $c_a^{(0)} \in \mathcal{C}$ where $a = f(g_1(*), \ldots, g_n(*))$.

Generating Clones

To get something more interesting, we need to consider clones that are generated by

- certain basic functions $\mathcal{F}$, and/or
- closed under additional operations Op.

We write

$$\mathsf{clone}(\mathcal{F}; \mathsf{Op})$$

for the least clone containing $\mathcal{F}$ and closed under Op.

For example, clone(;) consists just of all projections.

Rectypes

This is a perfect example of a recursive data type (rectype), one of the fundamental concepts in TCS. We have

- a collection of atoms (indecomposable items), and
- a collection of constructors that can be applied to build more complicated, decomposable objects.

Because of this inductive structure we can perform inductive arguments, both to establish properties and to define operations.

Basic Arithmetic Functions

When dealing with natural numbers, it is natural (duh) to have

- Constant zero 0
- Successor function $S : \mathbb{N} \to \mathbb{N},\ S(x) = x + 1$

Here constant 0 is meant to be the hard constant $c_0^{(0)}$ (but recall the comment on nullary composition from above).

This is a rather spartan set of built-in functions, but as we will see it's all we need. Needless to say, these functions are trivially computable.

In fact, it is hard to give a reasonable description of the natural numbers without them (unless you are a set theorist).

Closure Operations: Primitive Recursion

Given $h : \mathbb{N}^{n+2} \to \mathbb{N}$ and $g : \mathbb{N}^n \to \mathbb{N}$ we define a new function $f : \mathbb{N}^{n+1} \to \mathbb{N}$ by

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

Write $\mathrm{Prec}[h, g]$ for this function.

Definition

A function is primitive recursive (p.r.) if it lies in the clone generated by zero, successor; and closed under primitive recursion: $\mathrm{clone}(0, S; \mathrm{Prec})$.

Two Views

	bureaucracy	basic	operator
atom	projections	zero, successor	-
constructors	composition	-	prim. rec.

Example: Factorials

The standard definition of the factorial function uses recursion like so:

$$f(0) = 1$$
$$f(x+1) = (x+1) \cdot f(x)$$

To write the factorial function in the form $f = \mathrm{Prec}[h, g]$ we need

$$g : \mathbb{N}^0 \to \mathbb{N}, \quad g() = 1$$
$$h : \mathbb{N}^2 \to \mathbb{N}, \quad h(u, v) = (u+1) \cdot v$$

g is none other than $S \circ 0$ and h is multiplication combined with the successor function:

$$f = \mathrm{Prec}[\mathsf{mult} \circ (S \circ P_1^2, P_2^2), S \circ 0]$$

Digging Down

To get multiplication we use another recursion:

$$\mathsf{mult}(0, y) = 0$$
$$\mathsf{mult}(x+1, y) = \mathsf{add}(\mathsf{mult}(x, y), y)$$

Note that the 0 here technically stands for the unary version $c_0^{(1)}$ to conform with the specs for primitive recursion.

This is not an issue, since we know that our clone contains all $c_a^{(n)}$, $n \geq 0$.

And Addition?

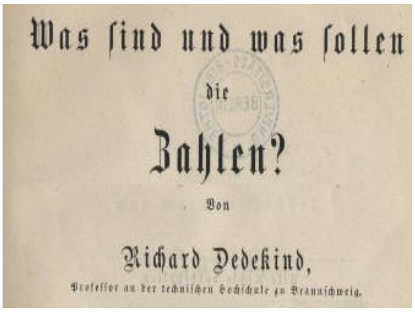
We have used addition, which can in turn be defined by yet another recursion.

$$\mathsf{add}(0, y) = y$$
$$\mathsf{add}(x+1, y) = S(\mathsf{add}(x, y))$$

We write x and y for simplicity, we really should use projections.

Since S is a basic function, we now have a complete, inductive proof that factorial is primitive recursive. And a way to compute factorials (at least in principle).

R. Dedekind



These equational, inductive definitions of basic arithmetic functions date back to Dedekind's 1888 paper "Was sind und was sollen die Zahlen?"

Arithmetic

It is a good idea to go through the definitions of all the standard basic arithmetic functions from the p.r. point of view.

$$\begin{aligned} \mathsf{add} &= \mathsf{Prec}[\mathsf{S} \circ \mathsf{P}^3_2, \mathsf{P}^1_1] \\ \mathsf{mult} &= \mathsf{Prec}[\mathsf{add} \circ (\mathsf{P}^3_2, \mathsf{P}^3_3), 0] \\ \mathsf{pred} &= \mathsf{Prec}[\mathsf{P}^2_1, 0] \\ \mathsf{sub}' &= \mathsf{Prec}[\mathsf{pred} \circ \mathsf{P}^3_2, \mathsf{P}^1_1] \\ \mathsf{sub} &= \mathsf{sub}' \circ (\mathsf{P}^2_2, \mathsf{P}^2_1) \end{aligned}$$

Since we are dealing with $\mathbb{N}$ rather than $\mathbb{Z}$, sub here is proper subtraction: $x \dot{-} y = x - y$ whenever $x \geq y$, and 0 otherwise.

Exercise

Show that all these functions behave as expected.

Informal versus Formal

Strictly speaking, in order to exhibit a p.r. function, we should write down a term in the corresponding programming language. For example, the following expression shows that the factorial function is p.r.

$$\mathsf{Prec}[\mathsf{Prec}[\mathsf{Prec}[\mathsf{S} \circ \mathsf{P}^3_2, \mathsf{P}^1_1] \circ (\mathsf{P}^3_2, \mathsf{P}^3_3), 0] \circ (\mathsf{S} \circ \mathsf{P}^2_1, \mathsf{P}^2_2), 1]$$

where we have written 1 for $S \circ 0$ for legibility. The innermost Prec yields addition, the next multiplication and the last factorial.

This is an instance of the old battle between formal and informal proofs. If you are a theorem prover, the formal version is far better. But it is very hard on the human eye: we will usually prefer the informal descriptions from above.

A Programming Language

We could think of our system as a simple programming language PR for arithmetic functions. The "programs" are just well-formed terms using the basic ingredients: projections, zero, successor, composition and primitive recursion.

Any such term τ describes an arithmetic function $\tau^\star : \mathbb{N}^k \to \mathbb{N}$.

Of course, RP programs also form a rectype: atoms are projections, zero and successor; constructors are composition and primitive recursion.

Evaluation

Based on RP being a rectype, it would be quite straightforward to program out an evaluation operator eval that takes as input any well-formed term τ of arity n and input $\boldsymbol{x} = x_1, \ldots, x_n \in \mathbb{N}$:

$$\mathsf{eval}(\tau, \boldsymbol{x}) = \text{ value of } \tau^\star \text{ on arguments } \boldsymbol{x}$$

Exercise

Write a compiler that given any string τ checks whether it is a well-formed expression denoting a primitive recursive function.

Exercise

Write an interpreter for primitive recursive functions (i.e., implement eval*) in your favorite programming language.*

A Primitive Recursive Zoo

We have seen that basic arithmetic functions such as addition, multiplication and proper subtraction are all primitive recursive.

In fact, it is quite difficult to come up with an arithmetic function that fails to be primitive recursive, yet is somehow intuitively computable. Go through any basic book on number theory, everything will be p.r.

To show that lots of functions are primitive recursive we need two tools:

- A pool of known p.r. functions, and
- strong closure properties.

A Slog

The results in this section are very technical, we have to formally verify that certain operations on functions do not affect primitive recursiveness.

Once you have gone through the technical details once, try to ignore them and focus on developing intuition that explains why a function is primitive recursive (rather than just proving it by writing down some incomprehensible formal expression).

Admissibility

Here is an example of a closure property that is not obvious from the definitions. Apparently, we lack a mechanism for definition-by-cases:

$$f(x) = \begin{cases} 3 & \text{if } x < 5, \\ x^2 & \text{otherwise.} \end{cases}$$

We know that $x \mapsto 3$ and $x \mapsto x^2$ are p.r., but is f also p.r.?

We want to show that definition by cases is admissible in the sense that when applied to primitive recursive functions/relations we obtain another primitive recursive function. Having lots of admissible operations around makes it easier to show that some functions are primitive recursive.

Definition by Cases

Definition

Let $g, h : \mathbb{N}^n \to \mathbb{N}$ and $R \subseteq \mathbb{N}^n$.
Define $f = \mathsf{DC}[g, h, R]$ by

$$f(\boldsymbol{x}) = \begin{cases} g(\boldsymbol{x}) & \text{if } \boldsymbol{x} \in R, \\ h(\boldsymbol{x}) & \text{otherwise.} \end{cases}$$

We need to explain what it means for the relation R to be primitive recursive, we'll do that in a minute.

Sign and Inverted Sign

The first step towards implementing definition-by-cases is a bit strange, but we will see that the next function is actually quite useful.

The sign function is defined by

$$\mathsf{sign}(x) = \min(1, x)$$

so that $\mathsf{sign}(0) = 0$ and $\mathsf{sign}(x) = 1$ for all $x \geq 1$. Sign is primitive recursive: $\mathsf{Prec}[\mathsf{S} \circ 0, 0]$ in sloppy notation.

Similarly the inverted sign function is primitive recursive:

$$\overline{\mathsf{sign}}(x) = 1 \dot{-} \mathsf{sign}(x)$$

Relations

As usual, define the characteristic function of a relation R

$$\mathsf{char}_R(\boldsymbol{x}) = \begin{cases} 1 & \boldsymbol{x} \in R \\ 0 & \text{otherwise.} \end{cases}$$

to translate relations into functions.

Definition

A relation is primitive recursive if its characteristic function is primitive recursive.

We will use analogous definitions later for all kinds of other types of computable functions: Turing, polynomial time, polynomial space, whatever.

Equality and Order

Define $E : \mathbb{N}^2 \to \mathbb{N}$ by

$$E = \overline{\mathsf{sign}} \circ \mathsf{add} \circ (\mathsf{sub} \circ (\mathsf{P}^2_1, \mathsf{P}^2_2), \mathsf{sub} \circ (\mathsf{P}^2_2, \mathsf{P}^2_1))$$

Or, less formally, but more intelligible:

$$E(x, y) = \overline{\mathsf{sign}}((x \dot{-} y) + (y \dot{-} x))$$

Then $E(x, y) = 1$ iff $x = y$, and 0 otherwise. Hence equality is primitive recursive. Even better, all standard order relations such as

$$\neq,\ \leq,\ <,\ \geq, \dots$$

are primitive recursive (so we can use them e.g. in definitions by cases).

Closure Properties

Proposition

The primitive recursive relations are closed under intersection, union and complement.

Proof.

$$\begin{aligned}\mathsf{char}_{R\cap S} &= \mathsf{mult} \circ (\mathsf{char}_R, \mathsf{char}_S)\\ \mathsf{char}_{R\cup S} &= \mathsf{sign} \circ \mathsf{add} \circ (\mathsf{char}_R, \mathsf{char}_S)\\ \mathsf{char}_{\mathbb{N}-R} &= \mathsf{sub} \circ (\mathsf{S} \circ 0, \mathsf{char}_R)\end{aligned}$$

□

In other words, primitive recursive relations form a Boolean algebra, and even an effective one: we can compute the Boolean operations.

Arithmetic and Logic

Note what is really going on here: we are using arithmetic to express logical concepts such as disjunction.

The fact that this translation is possible, and requires very little on the side of arithmetic, is a central reason for the algorithmic difficulty of many arithmetic problems: logic is hard, by implication arithmetic is also difficult.

For example, finding solutions of Diophantine equations is hard.

Exercise

Show that every finite set is primitive recursive. Show that the even numbers are primitive recursive.

DC is Admissible

Proposition

If g, h, R are primitive recursive, then $f = \mathsf{DC}[g, h, R]$ is also primitive recursive.

Proof.

$$f = \mathsf{add} \circ (\mathsf{mult} \circ (\mathsf{char}_R, g), \mathsf{mult} \circ (\overline{\mathsf{char}}_R, h))$$

Less cryptically

$$f(\boldsymbol{x}) = \mathsf{char}_R(\boldsymbol{x}) \cdot g(\boldsymbol{x}) + \overline{\mathsf{char}}_R(\boldsymbol{x}) \cdot h(\boldsymbol{x})$$

Since either $\mathsf{char}_R(\boldsymbol{x}) = 0$ and $\overline{\mathsf{char}}_R(\boldsymbol{x}) = 1$, or the other way around, we get the desired behavior. □

Bounded Sum

Proposition

Let $g : \mathbb{N}^{n+1} \to \mathbb{N}$ be primitive recursive, and define

$$f(x, \boldsymbol{y}) = \Sigma_{z<x} g(z, \boldsymbol{y})$$

Then $f : \mathbb{N}^{n+1} \to \mathbb{N}$ is again primitive recursive. The same holds for products.

Proof.

$$f = \mathsf{Prec}[\mathsf{add} \circ (g \circ (P^{n+2}_1, P^{n+2}_3, \dots, P^{n+2}_{n+2}), P^{n+2}_2), 0^n]$$

Less formally,

$$\begin{aligned}f(0, \boldsymbol{y}) &= 0\\ f(x^+, \boldsymbol{y}) &= f(x, \boldsymbol{y}) + g(x, \boldsymbol{y})\end{aligned}$$

Here we have written x^+ instead of $x + 1$. Yes, that helps.

Exercises

Exercise

Repeat the proof for products.

Exercise

Show that $f(x, \boldsymbol{y}) = \sum\big(g(z, \boldsymbol{y}) \mid z < x \wedge R(z)\big)$ is primitive recursive when g and R are primitive recursive.

Exercise

Show that $f(x, \boldsymbol{y}) = \sum_{z<h(x)} g(z, \boldsymbol{y})$ is primitive recursive when h is primitive recursive and strictly monotonic. How about a general function h?

Bounded Search

A particularly important algorithmic technique is search over some finite domain.

For example, in brute-force factoring n we are searching over an interval $[2, n-1]$ for a number that divides n. Or in a chess program we search for the optimal next move over a space of possible next moves.

We can model search in the realm of p.r. functions as follows.

Definition (Bounded Search)

Let $g : \mathbb{N}^{n+1} \to \mathbb{N}$. Then $f = \mathsf{BS}[g] : \mathbb{N}^{n+1} \to \mathbb{N}$ is the function defined by

$$f(x, \boldsymbol{y}) = \begin{cases} \min\big(\, z < x \mid g(z, \boldsymbol{y}) = 0 \,\big) & \text{if } z \text{ exists,} \\ x & \text{otherwise.} \end{cases}$$

Keeping Things Simple

Note that $f(x, \boldsymbol{y}) = x$ simply means that the search failed. In a more luxurious environment we might set a flag, throw an exception or some such.

Here we want everything to be a simple as possible, and in particular constrained to pure arithmetic. So we code failure as a numerical value, basta.

BS is Admissible

One can show that bounded search is also admissible, it adds nothing to the class of p.r. functions.

Proposition

If g is primitive recursive, then so is $\mathsf{BS}[g]$.

Exercise

Show that bounded search is indeed admissible ("primitive recursive functions are closed under bounded search").

Bounded Search II

This can be pushed a little further: the search does not have to end at x. Instead, we can search up to a primitive recursive function of x and $\boldsymbol{y}$.

$$f(x, \boldsymbol{y}) = \begin{cases} \min\big(\, z < h(x, \boldsymbol{y}) \mid g(z, \boldsymbol{y}) = 0 \,\big) & \text{if } z \text{ exists,} \\ h(x, \boldsymbol{y}) & \text{otherwise.} \end{cases}$$

Dire Warning:

But we have to have a p.r. bound, unbounded search as in

$$f(\boldsymbol{y}) := \min\big(\, z \mid g(z, \boldsymbol{y}) = 0 \,\big)$$

is not an admissible operation; not even when there is a suitable witness z for each $\boldsymbol{y}$. See the discussion of the μ-operator later.

Example: Primality

Claim (1)

The divisibility relation $\mathrm{div}(x, y)$ *is primitive recursive.*

Note that

$$\mathrm{div}(x, y) \iff \exists\, z \leq y\, (x * z = y)$$

so that bounded search intuitively should suffice to obtain divisibility. Formally, we have already seen that the characteristic function $M(z, x, y)$ of $x * z = y$ is p.r. But then

$$\mathsf{sign}\left(\sum\nolimits_{z \leq y} M(z, x, y)\right)$$

is the p.r. characteristic function of div.

Primality

Claim (2)

The primality relation is primitive recursive.

To see why, note that x is prime iff

$$1 < x \wedge \forall\, z < x\, (\mathrm{div}(z, x) \Rightarrow z = 1).$$

The building blocks $1 < x$, div and $z = 1$ are all p.r., and we can combine things by $\wedge$ and $\Rightarrow$. The only potential problem is the (bounded) universal quantifier.

But this is quite similar to the situation with div from the last slide. Time for a general solution.

Yet More Logic

Arguments like the ones for basic number theory suggest another type of closure properties, with a more logical flavor.

Definition (Bounded Quantifiers)

$P_\forall(x, \boldsymbol{y}) \Leftrightarrow \forall\, z < x\, P(z, x, \boldsymbol{y})$ and $P_\exists(x, \boldsymbol{y}) \Leftrightarrow \exists\, z < x\, P(z, x, \boldsymbol{y})$.

Note that $P_\forall(0, \boldsymbol{y}) = \text{true}$ and $P_\exists(0, \boldsymbol{y}) = \text{false}$.

Informally, and using the dreaded ellipsis,

$$P_\forall(x, \boldsymbol{y}) \iff P(0, x, \boldsymbol{y}) \wedge P(1, x, \boldsymbol{y}) \wedge \ldots \wedge P(x-1, x, \boldsymbol{y})$$

and likewise for $P_\exists$.

Bounded Quantification

Bounded quantification is really just a special case of bounded search: for $P_\exists(x, \boldsymbol{y})$ we search for a witness $z < x$ such that $P(z, x, \boldsymbol{y})$ holds. Generalizes to $\exists\, z < h(x, \boldsymbol{y})\, P(z, x, \boldsymbol{y})$ and $\forall\, z < h(x, \boldsymbol{y})\, P(z, x, \boldsymbol{y})$.

Proposition

Primitive recursive relations are closed under bounded quantification.

Proof.

$$\text{char}_{P_\forall}(x, \boldsymbol{y}) = \prod_{z<x} \text{char}_P(z, x, \boldsymbol{y})$$

$$\text{char}_{P_\exists}(x, \boldsymbol{y}) = \text{sign}\left(\sum_{z<x} \text{char}_P(z, x, \boldsymbol{y})\right)$$

□

Next Prime

Claim (3)

The next prime function $f(x) = \min(\, z > x \mid z \text{ prime}\,)$ is p.r.

This follows from the fact that we can bound the search for the next prime by a p.r. function:

$$f(x) \leq 2x \qquad \text{for } x \geq 1.$$

This bounding argument requires a little number theory. In general, the theory of gaps between consecutive primes is quite difficult (consider prime twins), but this result is not too bad.

Enumerating Primes

Claim (4)

The function $n \mapsto p_n$, where p_n is the nth prime, is primitive recursive.

To see this we can iterate the "next prime" function from the last claim:

$$p(0) = 2$$
$$p(n+1) = f(p(n))$$

Exercises

Exercise

Show in detail that the function $n \mapsto p_n$ where p_n is the nth prime is primitive recursive. How large is the p.r. expression defining the function?

Exercise

Find some other closure properties of primitive recursive functions.

Faking Data structures

Our primitive recursive programming language has one glaring defect: it only supports one data type, $\mathbb{N}$. There are no lists, trees, graphs, hash tables and so on, only natural numbers.

As it turns out, all these discrete structures can be obtained from just integers if we are able to express sequences $a_0, a_1, \ldots, a_{n-1}$ of numbers as a single number $\langle a_0, a_1, a_2, \ldots, a_{n-1} \rangle$.

This is obviously not meant as a practical programming idea, it is purely conceptual: natural numbers already suffice in principle, and the ability to compute with them means that other computation involving, say, list, are also possible.

Leopold Kronecker, Semi-Constructivist

Die ganzen Zahlen hat der liebe Gott gemacht, alles andere ist Menschenwerk.

"Dear god" made the integers, everything else is the work of men.

Coding

Write $\mathbb{N}^\star$ for the set of all finite sequences of natural numbers and nil for the empty sequence.

We would like to express a sequence $a_0, a_1, \ldots, a_{n-1} \in \mathbb{N}^\star$ as a single number $\langle a_0, a_1, \ldots, a_{n-1} \rangle$. So we need a coding function, a polyadic map of the form

$$\langle . \rangle : \mathbb{N}^\star \to \mathbb{N}$$

that allows us to decode: from $b = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ we can recover n as well as all the a_i.

Note that the coding function must necessarily be injective. Moreover, both the coding and decoding operations should be computationally cheap, at least primitive recursive.

Decoding

Suppose

$$b = \langle a_0, a_1, a_2, \ldots, a_{n-1} \rangle$$

is some code number. Note that we have used 0-indexing to simplify notation below.

We want a unary length function $\mathsf{len} : \mathbb{N} \to \mathbb{N}$ that determines the length of the coded sequence

$$\mathsf{len}(b) = n$$

and a binary decoding function $\mathsf{dec} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ that extracts the components:

$$\mathsf{dec}(b, i) = a_i$$

for all $i = 0, \ldots, n-1$. Traditionally, $\mathsf{dec}(b, i)$ is written $(b)_i$.

Computable Coding

Again, we need three functions:

$$\langle . \rangle : \mathbb{N}^\star \to \mathbb{N}$$

$$\mathsf{dec} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

$$\mathsf{len} : \mathbb{N} \to \mathbb{N}$$

In the set-theoretic universe, the existence of these functions is entirely trivial: $\mathbb{N}^\star$ is countable.

But we live in the computational universe: we need these functions to be easily computable, and in particular primitive recursive.

Sequence Numbers

The numbers of the form $\langle a_0, a_1, a_2, \ldots, a_{n-1} \rangle$ that appear as codes of sequences are called sequence numbers.

Note that a priori $\mathsf{len}(x)$ need not be defined when x is not a sequence number. The same is true for $\mathsf{dec}(x, i)$, plus i may be too large to make sense. Still, one usually insists that both decoding functions are total and return some default value like 0 for meaningless arguments.

Exercise

Show how to check if a number is a sequence number given dec *and* len.

Pairs

The first step is to select a pairing function, an injective map $\pi : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

Equivalently, we are looking for 3 functions $\pi : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, $\pi_i : \mathbb{N} \to \mathbb{N}$, $i = 1, 2$, such that

$$\pi_i(\pi(x_1, x_2)) = x_i$$

There are many possibilities, the following choice arguably yields one of the most intuitive coding functions.

$$\pi(x, y) = 2^x(2y + 1)$$

For example

$$\pi(5, 27) = 32 \cdot 55 = 1760 = 110111\,00000_2$$

In Binary

Note that the binary expansion of $\pi(x, y)$ looks like so:

$$y_k y_{k-1} \dots y_0\, 1\, \underbrace{00\dots0}_{x}$$

where $y_k y_{k-1} \dots y_0$ is the standard binary expansion of y (y_k is the most significant digit). Hence the range of π is $\mathbb{N}_+$ (but not $\mathbb{N}$).

This makes it easy to find the corresponding unpairing functions:

$$x = \pi_1(\pi(x, y)) \qquad y = \pi_2(\pi(x, y)).$$

Aside: Fueter-Pólya

Another popular pairing function is the quadratic polynomial due to Cantor:

$$p(x, y) = ((x + y)^2 + 3x + y)/2$$

Note that this function is a bijection (unlike our exponential pairing function which misses 0).

A surprising theorem by Fueter and Pólya from 1923 states that, up to a swap of variables, this is the only quadratic polynomial that defines a bijection $\mathbb{N}^2 \leftrightarrow \mathbb{N}$.

The proof is rather difficult and uses the fact that e^a is transcendental for algebraic $a \neq 0$.

It is an open problem whether there are other bijections for higher degree polynomials. Extra Credit.

Extending to Sequences

$$\langle \mathsf{nil} \rangle := 0$$
$$\langle a_0, \dots, a_{n-1} \rangle := \pi(a_0, \langle a_1, \dots, a_{n-1} \rangle)$$

Here are some sequence numbers for this particular coding function:

$$\langle 10 \rangle = 1024$$
$$\langle 0, 0, 0 \rangle = 7$$
$$\langle 1, 2, 3, 4, 5 \rangle = 532754$$

It's a Bijection

Lemma

$\langle . \rangle : \mathbb{N}^* \to \mathbb{N}$ is a bijection.

Proof. Suppose

$$\langle a_0, \dots, a_{n-1} \rangle = \langle b_0, \dots, b_{m-1} \rangle$$

We may safely assume $0 < n \leq m$ (why?).

Since π is a pairing function, we get $a_0 = b_0$ and $\langle a_1, \dots, a_{n-1} \rangle = \langle b_1, \dots, b_{m-1} \rangle$.

By induction, $a_i = b_i$ for all $i = 1, \dots, n-1$ and $0 = \langle \mathsf{nil} \rangle = \langle b_n, \dots, b_{m-1} \rangle$. Hence $n = m$ and our map is injective.

Exercise

Prove that the function is surjective.

Less formally . . .

Here is a sequence number and its binary expansion:

$$\begin{aligned}\langle 2, 3, 5, 1 \rangle &= 20548 \\ &= 1\underbrace{0}_{1}1\underbrace{00000}_{5}1\underbrace{000}_{3}1\underbrace{00}_{2}\end{aligned}$$

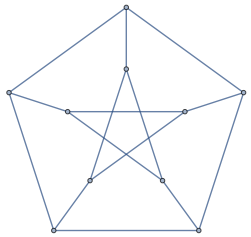
So the number of 1's (the digitsum) is just the length of the sequence, and the spacing between the 1's indicates the actual numerical values.

It follows that the coding function is injective and surjective, right?

That's It! 66

We can now code any discrete structure as an integer by expressing it as a nested list of natural numbers, and then applying the coding function.

For example, the so-called Petersen graph on the left is given by the nested list on the right.



$$((1,3),(1,4),(2,4),(2,5),(3,5),\\(6,7),(7,8),(8,9),(9,10),(6,10),\\(1,6),(2,7),(3,8),(4,9),(5,10))$$

Everything is PR 67

Exercise

Show that the pairing function π and both unpairing functions $x = \pi_1(\pi(x,y))$ and $y = \pi_2(\pi(x,y))$ are primitive recursive.

Exercise

Show that the length and decoding functions len *and* dec *are primitive recursive.*

Exercise

Show that the coding function $\langle . \rangle$ is primitive recursive when restricted to inputs of fixed length.

Recording History 68

One neat application of sequence numbers is course-of-value recursion. First note that ordinary primitive recursion can be expressed in terms of sequence numbers like so:

$$f(x,\boldsymbol{y}) = z \iff \exists\, s \in \mathsf{Seq}\,\big(\mathsf{len}(s) = x^+ \wedge (s)_0 = g(\boldsymbol{y}) \wedge \\ \forall\, 0 \le i < x\,((s)_{i^+} = h(i,(s)_i,\boldsymbol{y})) \wedge (s)_x = z\big)$$

Here x^+ is shorthand for $x+1$. The sequence number s records all previous values of f. Now consider the following function associated with f:

$$\widehat{f}(x,\boldsymbol{y}) := \langle f(0,\boldsymbol{y}), f(1,\boldsymbol{y}), \ldots, f(x,\boldsymbol{y})\rangle$$

Lemma

f is primitive recursive iff $\widehat{f}$ is primitive recursive.

Course of Value Recursion 69

Thus, it is natural to generalize the primitive recursion scheme slightly by defining functions so that the value at x depends directly on all the previous values.

$$f(0,\boldsymbol{y}) = g(\boldsymbol{y})$$
$$f(x^+,\boldsymbol{y}) = H(x,\widehat{f}(x,\boldsymbol{y}),\boldsymbol{y})$$

Lemma

If g and H are primitive recursive then f is also primitive recursive.

Exercise

Prove the last two lemmata. You may safely assume that standard sequence operations such as append are primitive recursive.

Observation 70

Claim

In the RealWorld™, every computable function is already primitive recursive.

This is, of course, nothing like a theorem, just a practical observation: natural computable functions have a very strong tendency to be primitive recursive.

All counterexamples to this rule are somehow "artificial."

Gödel's Approach

There is more elegant and slightly more elementary way to code sequence numbers due to Gödel that he used in his famous incompleteness theorem.

For the sake of completeness, here is a brief description of Gödel's method.

Gödel's Trick

To deal with sequences of arbitrary length one can use a clever divisibility argument.

Lemma (Gödel)

There exists a primitive recursive function $\mathrm{dec} : \mathbb{N}^2 \to \mathbb{N}$ *such that*

$$\forall\, a_0, \ldots, a_{n-1}\, \exists\, a\, \forall\, i < n\, (a_i = \mathrm{dec}(a, i)).$$

So a is a potential code number for $a_0, \ldots, a_{n-1}$

Proof. Set

$$\mathrm{dec}(a, i) = \min\big(x < a \mid \big((\pi(x, i) + 1)\pi_2(a) + 1\big) \text{ divides } \pi_1(a) \big)$$

The idea is that the factors of $\pi_1(a)$ contain information about the a_i.

We need to establish the existence of the witness a.

Proof, contd.

Let $a_0, \ldots, a_{n-1}$ arbitrary and set

$$\begin{aligned} c &= \max\big(\pi(a_i, i) \mid i < n \big) \\ C &= (c-1)! \\ p &= \prod_{i<n} ((\pi(a_i, i) + 1)C + 1) \\ a &= \pi(p, C) \end{aligned}$$

Note that $\forall\, i < j < c\, (iC + 1, jC + 1 \text{ coprime })$.

But then

$$\mathrm{dec}(a, i) = \min\big(x < a \mid (\pi(x, i) + 1)C + 1 \text{ divides } p \big)$$

□

Sequence Numbers

Definition

Define a coding function $\langle . \rangle$ by

$$\langle \boldsymbol{x} \rangle = \min\big(a \mid \mathrm{dec}(a, 0) = n \wedge \forall\, i \in [n]\, (\mathrm{dec}(a, i) = a_i) \big)$$

Also set $\mathrm{lh}(a) = \mathrm{dec}(a, 0)$ and $(a)_i := \mathrm{dec}(a, i)$.

Again, $\langle . \rangle$ is not primitive recursive, but we have:

- $\mathrm{Seq} = \{\, \langle \boldsymbol{x} \rangle \mid \boldsymbol{x} \in \mathbb{N}^* \,\} \subseteq \mathbb{N}$ is primitive recursive.
- The restriction to $\mathbb{N}^n$ is primitive recursive.
- dec is primitive recursive.

Exercise

Prove this claim in detail.

Sequence Operations

As always, having a data structure by itself is not particularly interesting, we need to be able to implement operations. In our case, one can show that the following operations on sequences are primitive recursive.

- head, tail
- concatenate
- reverse
- sort
- map
- sum, product

In fact, it would be quite difficult to come up with any example of an operation used in a real program that fails to be primitive recursive.

Algorithms in the RealWorld™

We claim that any algorithm you will ever see, outside of a class dealing directly with logic and computability, is always primitive recursive. And, in fact, trivially so.

There are two parts to this claim:

- All these algorithms operate on finitary data structures that can be coded naturally as sequence numbers, and
- given this natural coding, for input as well as output, the corresponding functions are always primitive recursive.

Of course, there is no actual theorem here, just an observation. I'd be most curious to hear about anything that might contradict this claim.

PRec Bijections

Theorem (Kuznecov 1950)

The collection of bijective primitive recursive functions is not closed under inverse.

Proof. Define the Ackermann-like function

$$\begin{aligned} B_0(x) &= 2x \\ B_{n^+}(x) &= B_n^x(1) \\ B(x) &= B_x(x) \end{aligned}$$

It follows from monotonicity that the predicate "$B_n(x) = y$" is primitive recursive, uniformly in n, x, y.

Contd.

Let R be the range of $B : \mathbb{N} \to \mathbb{N}$, so R is infinite, co-infinite and primitive recursive. Note that R is very sparse.

Let π_X be the Hauptfunktion of $X \subseteq \mathbb{N}$ and define $f : \mathbb{N} \to \mathbb{N}$

$$f(x) = \begin{cases} 2\pi_R^{-1}(x) & \text{if } x \in R, \\ 2\pi_{\overline{R}}^{-1}(x) + 1 & \text{otherwise.} \end{cases}$$

Then f is an primitive recursive bijection, but f^{-1} fails to be primitive recursive.

Exercises

Exercise

Prove that all these functions are indeed primitive recursive.

Exercise

Explain how to implement search in binary search trees as a primitive recursive operation.

Exercise

Come up with yet another coding function based on repeated application of a pairing function (make sure your method really works).