CDM

Primitive Recursion

Klaus Sutner

Carnegie Mellon University

Fall 2024

Defining Computability

We need a rigorous definition of computability that is easy to understand and apply, and that matches our intuitive, pre-theoretic notion of computability.

Roughly speaking, there are two types of definitions that can be used:

- **Machine Models**
  Abstract, mathematical machines that capture the notion of a "computer" in a more or less physical sense.
- **Programms**
  A sequence of primitive instructions that can be executed in a simple, mechanical manner.

Easy!

Anyone who has ever written and executed a program will probably suggest a quick-and-dirty definition along the lines of

> Computable means: can be done, in principle, by a standard digital computer. One writes a program in some language and compiles/executes it on suitable hardware.

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?

How About Algorithms?

> Computable means: there is an algorithm for it.

Well, then what is an algorithm?

Sadly, there currenly is no good definition of an algorithm, even though the notion seems entirely obvious to any CS person. The usual attempts at an explanation come down to: an algorithm implements a computable function. Hopelessly circular and useless.

Here is a slightly more useful attempt, in the opposite direction:

> An algorithm is a computable function, taking into account implementation issues and resource bounds.

CS vs Math

It is noteworthy that, traditionally, 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. Consider any classical decidability proof (say, Tarski's proof of the decidability of the first-order theory of the reals) and you will see it is a lightyear away from an algorithm.

If you want to get a first impression of how one might go about developing a real and useful theory of algorithms, take a look at

Moschovakis 02

It's quite tricky to come up with something compelling.

Why Bother?

Why would we even need a rigorous definition of a concept that is so utterly intuitive? Isn't this all plain obvious?

Yes and No, but mostly: No.

Here is a heuristic argument: in the 1930s there was some tension between Church and Gödel about the proper notion of computability, the issue was finally resolved only with Turing's seminal paper.

If things were not "obvious" to these giants, they are indeed not obvious.

An informal approach is often good enough for positive results: such-and-such a thing is computable, just do this, then that, . . . , voila, the result. For example, the age-old Euclidean algorithm clearly describes a computable function, no theory needed. The same is true for arithmetic in general.

But things get treacherous when it comes to computation in classical math, in particular over the reals: a calculation might require to evaluate an integral in some infinite-dimensional space (very popular in physics). It is far from clear what exactly is going on from the perspective of computability—there currently is no compelling theory of computation in the continuous setting.

Non-Computability

Negative results absolutely depend on real foundations: in order to show that a particular problem (say, solving Diophantine equations) fails to be computable, we need to have an airtight definition of computability.

Things get much worse when we try to show that solving a particular computable problem (say, satisfiability testing for Boolen formulae) requires such and such resources, the key concern in complexity theory.

As we now understand, the latter type of question appears to be breathtakingly more difficult than a simple distinction between computable and non-computable.

Models of Computation

- K. Gödel: primitive recursive
- A. Church: λ-calculus
- J. Herbrand, K. Gödel: general recursiveness
- A. Turing: Turing machines
- S. C. Kleene: μ-recursive functions
- E. Post: production systems
- H. Wang: Wang machines
- A. A. Markov: Markov algorithms
- M. Minsky; J. C. Shepherdson, H. E. Sturgis: register machines
- Z. Manna: while programs

Comments

The models are listed roughly in historical order. Except for primitive recursive functions, these models are all equivalent in a strict technical sense.

This does **not** mean that they are equally intuitive or compelling. For example, unless you have the theory-gene, you will find the λ-calculus pretty daunting.

Bad news: the second most daunting model is Turing machines. They have a beautiful motivation and are very natural in a way, but when it comes to technical details they are a nightmare.

Alas, for complexity theory there is no way around Turing machines. Since you are already familiar with them, we will talk use register machines to get half-way serious about universality.

Computable Arithmetic Functions

To simplify matters a bit, initially 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 will be very familiar to anyone familiar with 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 x, $f(x, \boldsymbol{y})$ and $\boldsymbol{y}$.

You have all seen the standard examples: addition, multiplication, exponentiation, factorials and so on. As we will see shortly, it is quite difficult to come up with an arithmetic function that is intuitively computable but not primitive recursive.

Later we will see more complicated forms of recursion.

Details

If one cares about actual implementations of primitive recursive functions, there are two basic choices.

Top-Down Implement a recursion stack so that a call to $f(n, \boldsymbol{y})$ automatically produces calls to $f(n-1, \boldsymbol{y})$, $f(n-2, \boldsymbol{y})$ ... and handles the returns properly.

Bottom-Up Compute $f(0, \boldsymbol{y})$, $f(1, \boldsymbol{y})$, $f(2, \boldsymbol{y})$ ... by iteration, which requires only a simple loop.

Again, in math this distinction does not matter much, in the early days of CS it produced some acrimonious debates[†].

[†] There were fierce fights about whether Algol 60 should include recursion. I studied with one of the people on the wrong side of the argument.

Mental Health Warning

One can describe primitive recursive function simply as all arithmetic functions that can be generated from 0, successor and projections using composition and primitive recursion.

Add a few more words about what exactly we mean by these various ingredients and we're done. Anyone with a suitable background will understand exactly what is meant.

All true, but if we wanted an actual implementation of primitive recursive functions there are lots of missing details. To build an interpreter one has to work a bit harder and pin down lots of details that are usually left out.

Think of the following as a proof-of-concept. In the future we will be far more casual about definitions.

Details: Primitive Recursive Functions

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

The notion of "recursive function" today refers to an arbitrary computable function. The key difference is that primitive recursive functions can only use recursion on one variable, whereas full computability requires recursion on multiple variables as in the Herbrand-Gödel model of computation.

For primitive recursive functions it will always be crystal clear that they 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 $h \circ (g_1, \ldots, g_n)$ inspired by the the well-known special case $m = 1$:

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

So this is just ordinary sequential composition of functions. Clearly, computable functions are closed wrto composition: output can be re-used as input.

Primitive Recursion

We need one more operation beyond composition to get interesting functions, a form of 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

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

Here is our preliminary definition.

Definition

A function is primitive recursive (p.r.) if it can be generated from zero, successor, composition and primitive recursion.

Arithmetic Functions

All the basic functions of arithmetic are primitive recursive.

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

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

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

Summation

Suppose ℓ is a primitive recursive function and we want to sum its values. No problem:

$$f(0) = 0$$
$$f(x{+}1) = \mathsf{add}(f(x), \ell(x))$$

So $f(x) = \sum_{z<x} \ell(z)$.

Similarly, if ℓ has an additional parameter, we cad adjust the definition as follows:

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

No problem.

For a human reader, this is indeed all perfectly clear.

But there is a minor issue: the two zeros are different.

$$f(0) = 0$$
$$f(0, y) = 0$$

The first one has arity 0, but the second has arity 1. This is forced by our definition of primitive recursion.

A little more precision is needed if we wanted to, say, check proofs involving primitive recursive functions.

Arity

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 $\mathsf{C}_{\mathsf{a}}^{(\mathsf{n})}$ for the n-ary constant map $\boldsymbol{x} \mapsto a$.

We will call $\mathsf{C}_{\mathsf{a}}^{(0)}$ a hard constant: a function that takes no arguments.

Bureaucracy: Projections

Another problem is that composition as we defined it is not quite enough. Suppose we have a binary version add of addition, and want to define a ternary version. No problem:

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

But, this is **not** allowed according to our definition of composition: try to find the right binding for h and the g_i.

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

$$\mathsf{P}^{\mathsf{n}}_{\mathsf{i}} : \mathbb{N}^n \to \mathbb{N} \qquad \mathsf{P}^{\mathsf{n}}_{\mathsf{i}}(x_1, \ldots, x_n) = x_i$$

where $1 \leq i \leq n$.

Killing Variables

Now we can write

$$\mathsf{add}^{(3)} = \mathsf{add}^{(2)} \circ (\mathsf{P}^3_1, \mathsf{add}^{(2)} \circ (\mathsf{P}^3_2, \mathsf{P}^3_3))$$

Note that no variables are needed in this notation system.

Needless to say, most humans prefer the informal notation by a long shot. But then again, the last term is very easier to parse and evaluate.

Clones

A clone or function algebra is a collection of functions that contains all projections and is closed under composition, over some carrier set.

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.

Quoi?

Informally and intuitively, a primitive recursive function is obtained from zero, successor, composition and primitive recursion. Basta.

Projections and composition have nothing to do with the natural numbers, these concepts are perfectly general and apply to any domain.

On the other hand, 0, successor and primitive recursion are directly dependent on the naturals.

So it makes sense to separate out these two components of the definition of a primitive recursive function.

Nullary Functions

Note that we allow hard constants, nullary functions in $A^0 \to A$ where we think of A^0 as a one-point set $\{*\}$.

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

In the literature, you will also find clones without nullary 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???

Algebraists usually prefer the non-nullary approach. Most operations there are binary and unary: e.g., $(x, y) \mapsto x \cdot y$ and $x \mapsto x^{-1}$ in a group. Constants are just elements of the algebraic structure and are not considered to have anything to do with an operation.

But for those working in logic, type theory or category theory, nullary operations are not an issue at all. And, truth be told, any really solid implementation of primitive recursive functions also needs to keep track of all these gory details, otherwise things won't typecheck.

After all, a computer will not apply any algebraic common sense whatsoever, it will just follow the rules precisely as stated.

Nullary Composition

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

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

Case: $n = 0$

Then for $m \geq 1$ we have $\mathsf{C}^{(\mathsf{m})}_{\mathsf{h}(*)} \in \mathcal{C}$.

Case: $m = 0$

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

Another Angle

We could introduce constants $\mathsf{C}_{\mathsf{a}}^{(\mathsf{k})}$ for all a and k. Alas, that contradicts the basic principle of parsimony in axiomatization: use as few basic assumptions as possible. For example, if we have the successor function S, we can define $\mathsf{C}_{\mathsf{a}+1}^{(\mathsf{k})} = S \circ \mathsf{C}_{\mathsf{a}}^{(\mathsf{k})}$, so we only need $\mathsf{C}_{0}^{(\mathsf{k})}$.

We can use primitive recursion to deal with arity:

$$f(0, \boldsymbol{y}) = \mathsf{C}_{0}^{(\mathsf{k})}(\boldsymbol{y})$$
$$f(x{+}1, \boldsymbol{y}) = f(x, \boldsymbol{y})$$

This defines $\mathsf{C}_{0}^{(\mathsf{k}+1)}$ in terms of $\mathsf{C}_{0}^{(\mathsf{k})}$.

So all we really need is $\mathsf{C}_{0}^{(0)}$.

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

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

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

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

This is a perfect example of a recursive datatype (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 $0 : \mathbb{N}$
- Successor function $S : \mathbb{N} \to \mathbb{N}\,,\ S(x) = x + 1$

Here constant 0 is meant to be the hard constant $\mathsf{C}_0^{(0)}$ (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

We write Prec$[h, g]$ for primitive recursion: recall $h : \mathbb{N}^{n+2} \to \mathbb{N}$ and $g : \mathbb{N}^n \to \mathbb{N}$ can be used to define $f : \mathbb{N}^{n+1} \to \mathbb{N}$ by

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

Definition

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

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 $f = \mathsf{Prec}[h, g]$ we need

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

More precisely, g is $\mathsf{C}_1^{(0)}$ and h is multiplication combined with the successor function:

$$f = \mathsf{Prec}[\mathsf{mult} \circ (S \circ \mathsf{P}_1^2, \mathsf{P}_2^2), \mathsf{C}_1^{(0)}]$$

Unfolding

By unfolding the definition of mult we can write a single term in our language that defines the factorial function.

$$\mathsf{Prec}[\mathsf{Prec}[\mathsf{Prec}[\mathsf{S} \circ \mathsf{P}_2^3, \mathsf{P}_1^1] \circ (\mathsf{P}_2^3, \mathsf{P}_3^3), \mathsf{C}_0^{(1)}] \circ (\mathsf{S} \circ \mathsf{P}_1^2, \mathsf{P}_2^2), \mathsf{C}_1^{(0)}]$$

The innermost Prec yields addition, the next multiplication and the last, factorial.

Again, hard to read for a human, but perfectly suited for a parser. Given the right environment, it is not hard to build an interpreter for these terms.

Arithmetic

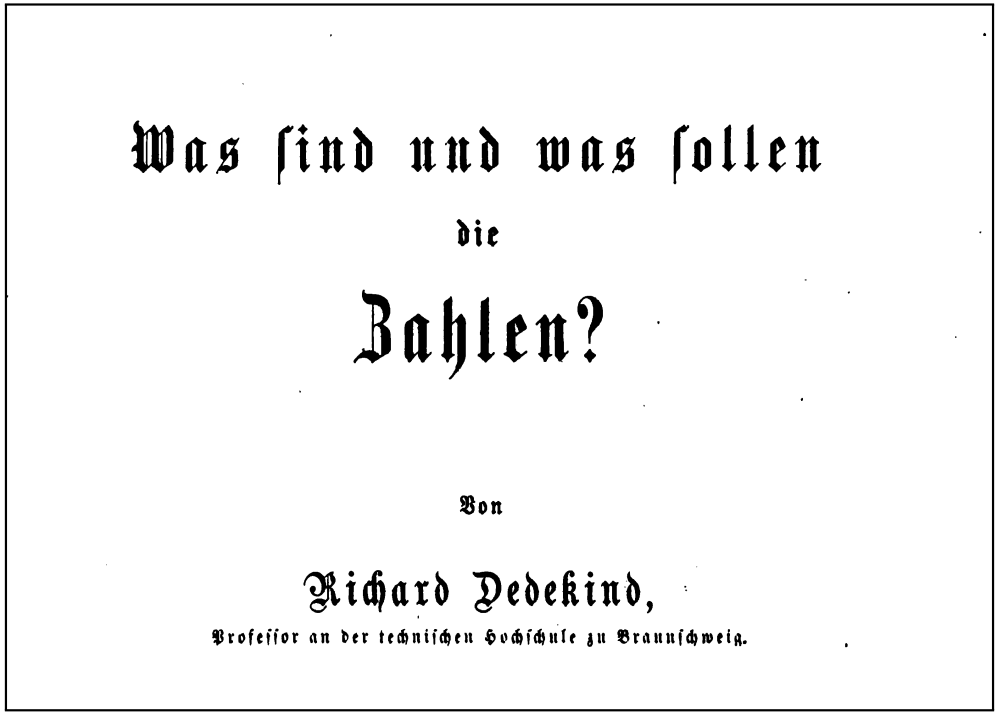
It is a good idea to go through the definitions of all the standard basic arithmetic functions from the p.r. point of view.

$$\begin{aligned}
\mathsf{add} &= \mathsf{Prec}[\mathsf{S} \circ \mathsf{P}^3_2, \mathsf{P}^1_1] \\
\mathsf{mult} &= \mathsf{Prec}[\mathsf{add} \circ (\mathsf{P}^3_2, \mathsf{P}^3_3), \mathsf{C}^{(1)}_0] \\
\mathsf{pred} &= \mathsf{Prec}[\mathsf{P}^2_1, \mathsf{C}^{(0)}_0] \\
\mathsf{sub}' &= \mathsf{Prec}[\mathsf{pred} \circ \mathsf{P}^3_2, \mathsf{P}^1_1] \\
\mathsf{sub} &= \mathsf{sub}' \circ (\mathsf{P}^2_2, \mathsf{P}^2_1)
\end{aligned}$$

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.



Was ſind und was ſollen
die
Zahlen?

Von

Richard Dedekind,
Profeſſor an der techniſchen Hochſchule zu Braunſchweig.

These equational, inductive definitions of basic arithmetic functions date back to Dedekind's 1888 booklet "What are numbers and what is their purpose?" It is remarkable that he produced this description about 30 years before anyone started to think carefully about computability.

Evaluation Operator

The terms in our little programming language form a rectype. In fact, this is a free rectype, we have unique decomposition.

Any term τ describes an arithmetic function $[\![+]\!]\tau : \mathbb{N}^k \to \mathbb{N}$ (the old intension vs extension story). There is a natural evaluation operator eval that takes as input any term τ of arity n and input $\boldsymbol{x} = x_1, \ldots, x_n \in \mathbb{N}$:

$$\text{eval}(\tau, \boldsymbol{x}) = \text{ value of } [\![+]\!]\tau \text{ on arguments } \boldsymbol{x}$$

As Gödel figured out, we can express the term τ as a natural number (its Gödel number), so that eval can be construed as an arithmetic function.

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 purpose of this section is to show in a fairly rigorous manner that certain operations on functions do not affect primitive recursiveness.

Once you have gone through the technical details, try to ignore them and focus on developing intuition that explains *why* a function is primitive recursive, rather than just slogging through the standard machinery of proof.

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 a solid collection of admissible operations around makes it much easier to show that some particular 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:

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

Relations

As usual, define the characteristic function of a relation R

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

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.

Digression: Sets and Functions

The idea that one can identify an arithmetic relation $R \subseteq \mathbb{N}^k$ with a function $\mathbb{N}^k \to \mathbf{2}$ may seem quaint and totally obvious.

Not so. In 1882 Cantor published his eponymous theorem, but instead of saying that the powerset has strictly larger cardinality he stated (in essence):

$$|S \to \mathbf{2}| > |S|$$

The collection of functions $S \to \mathbf{2}$ is just another way to describe the powerset of S.

Equality and Order

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

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

Or, less formally, but more intelligible:

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

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

$$\neq,\ \leq,\ <,\ \geq, \ldots$$

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

Proposition

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

Proof.

$$\begin{aligned}
\mathsf{char}_{R\cap S} &= \mathsf{mult} \circ (\mathsf{char}_R, \mathsf{char}_S) \\
\mathsf{char}_{R\cup S} &= \mathsf{sign} \circ \mathsf{add} \circ (\mathsf{char}_R, \mathsf{char}_S) \\
\mathsf{char}_{\mathbb{N}-R} &= \mathsf{sub} \circ (1, \mathsf{char}_R)
\end{aligned}$$

□

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

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.

$$f = \mathsf{Prec}[\mathsf{add} \circ (g \circ (P_1^{n+2}, P_3^{n+2}, \ldots, P_{n+2}^{n+2}), P_2^{n+2}), 0^n]$$

Less formally,

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

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. What if* h *is not monotonic?*

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 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 Kleene's μ-recursive functions.

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

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

is the p.r. characteristic function of div.

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.

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

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

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

□

Claim (3)

The next prime function $f(x) = \min\big(\, z > x \mid z \text{ prime} \big)$ *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:

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

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.

Primitive Recursive Bijections

As an example of a non-closure result we mention the following.

Theorem (Kuznecov 1950)

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

Sketch of 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 H_X be the principal function[†] of $X \subseteq \mathbb{N}$ and define $f : \mathbb{N} \to \mathbb{N}$

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

Then f is an primitive recursive bijection. Since B fails to be primitive recursive, f^{-1} is not.

□

[†]The function that enumerates the elements of X in order.

TM versus PR

Burning Question: How does the computational strength of Turing machines compare to primitive recursive functions?

Certainly, if Turing is right, TMs should be at least as powerful as p.r. functions.

It is a labor of love to check that indeed any p.r. function can be computed by a Turing machine. This comes down to building a TM compiler/interpreter for p.r. functions. Since we can use structural induction on the terms of our programming language this is quite straightforward given an environment that supports pattern matching (e.g., ML or Mathematica work well). On a Turing machine it is too tedious for consideration, but not hard in principle.

But how about the opposite direction?

In a Nutshell

Using the coding machinery described above it is not hard to see that the assertion "Turing machine M moves from ID C to ID C' in t steps" is primitive recursive (i.e., corresponds to a primitive recursive relation). This is not hard to show, just tedious.

But when we try to deal with "Turing machine M moves from ID C to ID C' in some number of steps" things fall apart: there is no obvious way to find a primitive recursive bound on the number of steps.

It is perfectly reasonable to conjecture that Turing computable is strictly stronger than primitive recursive, but coming up with a nice example is rather difficult.

Steps are p.r.

Proposition

Let M be a Turing machine. The t-step relation

$$C \vdash_M^t C'$$

is primitive recursive, uniformly in M.

Of course, this assumes a proper coding method for configurations and Turing machines.

For example, an instantaneous description is a triple $\langle T, i, p \rangle$ where $T : \mathbb{Z} \to \Sigma$, $i \in \mathbb{Z}$ and $p \in Q$.

Position i and state p are trivial to code as an integer. For the tape inscription T note that all but finitely values are blank, so we can simply list all the non-blank entries.

$$((i_1, a_1), (i_2, a_2), \ldots, (i_n, a_n))$$

which list can be coded by a single integer.

Hence we can encode a whole sequence of IDs

$$C = C_0, C_1, \ldots, C_{t-1}, C_t = C'$$

again by a single integer and check in a p.r. way that $C_i \left|\frac{1}{M}\right. C_{i+1}$.

A crucial ingredient here is that the size of the C_i is bounded by something like the size of C plus t, so we can bound the size of the sequence number coding the whole computation given just the size of C and t.

Exercise

Figure out exactly what is meant by the last comment.

Whole Computations?

Now suppose we want to push this argument further to deal with whole computations. We would like the transitive closure

$$C \left|\frac{*}{M}\right. C'$$

to be primitive recursive.

If we could bound the number of steps in the computation by some primitive recursive function of C then we could perform a brute-force search.

However, there is no obvious reason why such a bound should exist: the number of steps needed to get from C to C' could be enormous: recall the Marxen-Buntrock machine.

Likewise, there could well be no p.r. bound on the size of the required tape inscriptions, nor the head positions.

Again, there is a huge difference between bounded and unbounded search.

HG versus TM versus PR

Central Question: How does the computational strength of Turing machines compare to primitive recursive functions?

It is a labor of love to check that any p.r. function can indeed be computed by a TM.

This comes down to building a TM compiler/interpreter for p.r. functions. Since we can use structural induction this is not hard in principle; we can use a similar approach as in the construction of the universal TM.

Opposite Direction?

- The cheap answer is to point out that some TM-computable functions are not total, so they cannot be p.r.
- This is technically rue, but utterly boring. Here are the right questions:
- How much of a TM computation is primitive recursive?
- Is there a total TM-computable function that is not primitive recursive?

In a Nutshell

Using the coding machinery from last time it is not hard to see that the relation "TM M moves from configuration C to configuration C' in t steps" is primitive recursive.

But when we try to deal with "TM M moves from C to C' in some number of steps" things fall apart: there is no obvious way to find a primitive recursive bound on the number of steps.

It is perfectly reasonable to conjecture that TM-computable is strictly stronger than primitive recursive, but coming up with a nice example is rather difficult.

Steps are p.r.

Proposition

Let M be a Turing machine. The t-step relation

$$C \vdash_M^t C'$$

is primitive recursive, uniformly in t and M.

Of course, this assumes a proper coding method for configurations and Turing machines.

This is a straightforward application of the sequence numbers we discussed last week.

Likewise we can encode a whole sequence of configurations

$$C = C_0, C_1, \ldots, C_{t-1}, C_t = C'$$

again by a single integer.

And we can check in a p.r. way that $C \left|\frac{t}{M}\right. C'$.

A crucial ingredient here is that the size of the C_i is bounded by something like the size of C plus t, so we can bound the size of the sequence number coding the whole computation given just the size of C and t.

Exercise

Figure out exactly what is meant by the last comment.

Whole Computations?

Now suppose we want to push this argument further to deal with whole computations. We would like the transitive closure

$$C \vdash_M^* C'$$

to be primitive recursive.

If we could bound the number of steps in the computation by some p.r. function of C then we could perform a brute-force search.

However, there is no reason why such a bound should exist, the number of steps needed to get from C to C' could be enormous.

Again, there is a huge difference between bounded and unbounded search.