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.
- **Programs**
  A sequence of primitive instructions that can be executed in a simple, mechanical manner.

History Angle

- Classical Recursion theory (computability theory) started in the 1930s, well before the arrival of digital computers. It arose as a critical ingredient in any attempt to handle the Grundlagenkrise (foundational crisis) in mathematics early in the 20th century. CRT is not concerned with practical computation or applied algorithms, not at all.

- Complexity theory started in the 1950/60s, in response to the increasing relevance of actual digital computation, and the need to understand resource allocation issues (analysis of algorithms). This area is in part concerned with practical, realizable computation, but the theory part of complexity theory can also be far, far removed from real computation.

Grundlagenkrise

Why a crisis? Around 1900, some paradoxes such as Russell's self-contradictory set

$$S = \{ x \mid x \notin x \}$$

were casting doubt on the foundations of math. If everyone uses set theory as the foundation of math, and set theory is inconsistent, we have a little problem.

Key players: Hilbert, Herbrand, Gödel, Church, Kleene, Turing, Post.

In typical fashion, almost everybody simply ignored the issues and blissfully kept on doing math. What, me worry?

But some did pay attention, in particular David Hilbert.

Hilbert

Hilbert's Program

In the 1920s, partially in response to paradoxes and intuitionistic lunacy, David Hilbert[†] proposed a program to salvage all of mathematics. In a nutshell:

Formalize mathematics and concoct a finite set of axioms that are strong enough to prove all theorems of mathematics (completeness) and show that the system is free from contradictions (consistency); by strictly finitary means.

Also show that statements about "ideal objects" can be proven in the system, without using ideal objects.

[†]Full disclosure: Hilbert is my academic great-grandfather; of course, this plays no role whatsoever in my attitude towards his work :-)

Consistency and Completeness

Consistency here means that there is no formula Φ so that the system has a proof for Φ and also a proof for $\neg\Phi$.

We have an algorithm to check whether a string is a proof, but to check consistency it seems we need to run this algorithm infinitely often: check every possible proof.

Completeness means that, for every true formula Φ, the system has a proof for Φ.

In practice, this requirement could be softened a little bit, it's enough to have proofs for the statements one is actually interested in (a notion that is impossible to formalize).

And Computation?

Any reasonable formal system depends on computability: There is an algorithm to check whether

- a string is a valid formula
- a formula is an axiom
- a formula can be derived from one or two others using a rule of inference (e.g., modus ponens $A, A \Rightarrow B \vdash B$
- a sequence of deductive steps is a valid proof

In other words, a formal system amounts to so much wordprocessing[†].

[†] This may sound like an insult, but Hilbert was adamant that math, in principle, could be expressed by pushing symbols around on a piece of paper

Progress Surrounding Hilbert's Program

1889	Peano gives an axiomatization of basic number theory, using a Dedekind-style approach.
1899	Hilbert gives an axiomatization of basic geometry.
1910s	Russell and Whitehead build a fairly comprehensive formal system of mathematics based on types.
1918	Bernays shows that propositional logics is sound and complete.
1929	Gödel shows that first-order logic is sound and complete.

A Catastrophe

The Incompleteness Theorem was announced by Gödel on October 7, 1930, at a conference in Königsberg, the "First International Conference on the Philosophy of Mathematics":

> There is a formula of number theory such that Peano arithmetic proves neither the formula nor its negation.

Since either the formula or its negation must be true, we are missing out on a true statement of arithmetic.

It seems that the only person in the audience who understood what was going on was von Neumann. Hilbert was in Königsberg, he gave his "werden wissen" speech the next day, but apparently he did not attend Gödel's lecture.

Computation is Critical

One key idea in Gödel's proof is arithmetization[†]: one can express formulae and proofs, plus all necessary manipulations, in terms of arithmetic. Everything is coded up as a Gödel number.

The operations on Gödel numbers representing formulae, deduction steps and proofs are all easily computable.

Hence, they can be handled by any formal system that is is expressive enough to cover some basic aspects of arithmetic.

Note the irony: computability helps to demolish Hilbert's dream, which basically says everything is computable.

[†]Also useful in complexity theory.

Hilbert's Entscheidungsproblem

Another one of Hilbert's brilliant ideas.

> The Entscheidungsproblem is solved when one knows a procedure by which one can decide in a finite number of operations whether a given logical expression is generally valid or is satisfiable. The solution of the Entscheidungsproblem is of fundamental importance for the theory of all fields, the theorems of which are at all capable of logical development from finitely many axioms.
>
> D. Hilbert, W. Ackermann
> Grundzüge der theoretischen Logik, 1928

In modern terminology: Hilbert wanted a decision algorithm, more or less for all of mathematics. Again, the Entscheidungsproblem is directly based on computation.

No Luck

As we now know, the Entscheidungsproblem is unsolvable. However, in 1930, it was not known that this follows from Gödel's incompleteness theorem.

In order to **prove** that the Entscheidungsproblem is unsolvable one must have a solid theory of computation.

Note the asymmetry, to prove solvability of computational problems an informal argument is often quite enough.

As it turns out, the decision problem is highly non-computable even for basic arithmetic over $\mathbb{N}$. In fact, we cannot check whether a polynomial with integer coefficients has a solution over the integers (Matiyasevich).

One Century Later

Some quarters declare Hilbert's program to be dead. To me, that seems a major misconception; Hilbert had the right ideas, though they needed to be adjusted in various ways.

- His program jump-started the development of modern mathematical logic, in particular computability theory and proof theory. Both are critical for computer science.
- Finding decision algorithms for limited areas of discourse has been a very productive effort.
- Formalizing mathematics in Hilbert's days was completely hopeless. Today, in the presence of powerful computers, it is somewhat feasible and opens up endless possibilities.

Aside

From a modern perspective, the notion of computability may seem fairly straightforward. Do we really have to make a big fuss?

Things were far from obvious at the time. In fact, 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 at all.

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 as a model of computation.

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 acquainted with a modern programming language: recursion. For example, we can define the factorial function as follows:

$$\begin{aligned} f(0) &= 1 \\ f(x+1) &= (x+1) \cdot f(x) \end{aligned}$$

In essence, we are reducing the problem of computing factorials to computing products, plus some logical overhead.

It is intuitively clear from this description that f is computable.

Details

If one cares about the actual implementation of a recursive function, there are two basic choices.

Top-Down Implement a recursion stack so that a call to $f(n)$ automatically produces calls to $f(n-1)$, $f(n-2)$ $\ldots f(0)$ and handles the returns properly.

Bottom-Up Compute $f(0)$, $f(1)$, $f(2)$ $\ldots$, $f(n)$ 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

We are going to define primitive recursive functions in three different ways, first very informal, then semi-formal, essentially the standard of precision these days, and then very formal.

The third level is not necessary unless one really wants to get down to implementation issues and/or computer proofs.

We will never dig down that far again, I promise. In the future we will be far more casual about definitions.

Notation Warning

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.

Round 1: Informal

Definition (Informal)

An arithmetic function $f : \mathbb{N}^k \to \mathbb{N}$ is primitive recursive iff it can be generated from zero and successor using only the operations of composition and primitive recursion.

Think of this as describing a small programming language: zero and successor are built-in, and we can use two features in our language, composition and primitive recursion.

We can use these to build more complicated programs out of the basic ones. Nothing else is allowed.

Each of these programs defines a primitive recursive function.

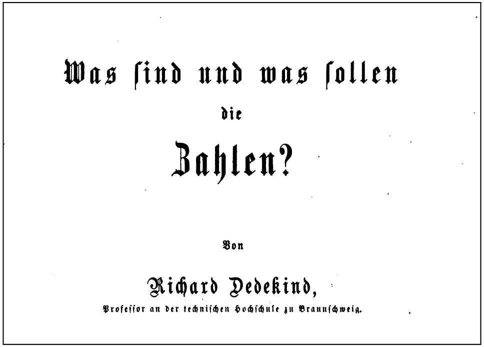
Arithmetic Functions

All the basic functions of arithmetic are primitive recursive according to this "definition."

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

In fact, it is quite hard to find an arithmetic function that is intuitively computable, but fails to be primitive recursive.

Aside: R. Dedekind



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.

Round 2: Semi-Formal

In round 1, we have appealed to common sense and a little background knowledge to figure out what composition and primitive recursion are.

Since both ideas are quite intuitive, there is not much harm in this.

But: if we want to show that some function fails to be primitive recursive, we better make sure to nail these down precisely.

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

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

Arities are critical here, otherwise things don't typecheck.

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.

Definition, Again

Since we have now explained what all the pieces mean, we can basically just repeat our first definition.

Definition (Semi-formal)

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

There, that’s it.

Or is it?

Round 3: Annoyingly Formal

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

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

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

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

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

No problem, everything matches our definition.

Pesky Constants

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

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

$$\begin{aligned} \mathsf{sum}(0) &= 0 \\ \mathsf{sum}(0, y) &= 0 \end{aligned}$$

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.

Bureaucracy: Projections

Another problem is that composition as we defined it is not quite enough. Suppose we have a binary version 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. This is nowadays called *tacit programming*, used in APL, J.

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

Abstraction

We are currently interested in the class of computable functions, but there are other collections that are also important; e.g., we might want to study functions that are polynomial time computable, or polynomial space computable.

It would pay off to figure out some characteristic properties of these and similar collections of functions.

Two basic properties stick out: projections are always included, and we want closure under composition.

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.

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 $\mathsf{C}_{\mathsf{h}(*)}^{(\mathsf{m})} \in \mathcal{C}$.

Case: $m = 0$

Then for $n \geq 1$ we have $\mathsf{C}_{\mathsf{a}}^{(0)} \in \mathcal{C}$ 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:

$$\begin{aligned} f(0, \boldsymbol{y}) &= \mathsf{C}_0^{(\mathsf{k})}(\boldsymbol{y}) \\ f(x{+}1, \boldsymbol{y}) &= f(x, \boldsymbol{y}) \end{aligned}$$

This defines $\mathsf{C}_0^{(\mathsf{k+1})}$ in terms of $\mathsf{C}_0^{(\mathsf{k})}$.

So all we really need is one hard constant: $\mathsf{C}_0^{(0)}$.

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 functional 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.

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).

Formal Definition

We write $\mathsf{Prec}[h, g]$ for primitive recursion as above.

Definition

A function is primitive recursive (p.r.) if it lies in the clone generated by the hard constant 0, the unary successor function; and is closed under primitive recursion: $\mathsf{clone}(\mathsf{C}_0^{(0)}, S^1; \mathsf{Prec})$.

A bit harder to read, but now there are no loose ends.

Example: Factorials

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

$$\begin{aligned} f(0) &= 1 \\ f(x+1) &= (x+1)\cdot f(x) \end{aligned}$$

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

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

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.

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

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, Again

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.

Aside: Rectypes

Primitive recursive functions (alternatively, the terms that define them) are 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.

*Rectype is a neologism that we have stolen from T. Forster in Cambridge; it is somewhat nonstandard, but it's too good not to use.

Two Views

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

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:

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

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.

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

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

Or, less formally, but more intelligible:

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

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

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

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.

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

□

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

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.

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

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. 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 $\mathsf{BS}[g]$.

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 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

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

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

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

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

□

Next Prime

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:

$$\begin{aligned} p(0) &= 2 \\ p(n+1) &= f(p(n)) \end{aligned}$$

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.

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.

$\square$

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