

CDM

Primitive and General Recursive Functions

Klaus Sutner
Carnegie Mellon University
www.cs.cmu.edu/~sutner

Battleplan

- Primitive Recursive Functions
- Bounded Search
- Admissible Operations
- Sequence Numbers
- Ackermann's Function
- (Partial) Recursive Functions
- Church-Turing Thesis
- Kleene Normal Form

Primitive Recursive Functions

Machines versus Programs

Machine models of computation are easy to describe and very natural. However, constructing any specific machine for a particular computation is rather tedious.

Almost always it is much preferable to use a *programming language* to explain how a computation should be performed.

There are lots of standard programming languages that all could be used to give alternative definitions of computability: Algol, Pascal, C, C++, Java, perl, . . .

The problem with all of these is that it is quite difficult to give a careful explanation of the semantics of the programs written in any of these languages.

Primitive Recursive Functions

To avoid problems with semantics we will introduce a language that has only one data type: \mathbb{N} , the non-negative integers.

Actually, before we describe the "programs" we will define the corresponding behaviors, the so-called primitive recursive functions, maps of the form

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

that can be computed intuitively using no more than a limited type of recursion.

We use induction to define this class of functions: we select a few simple basic functions and build up more complicated ones by composition and a limited type of recursion.

The Basic Functions

There are three types of basic functions.

- **Constants zero** $C^n : \mathbb{N}^n \rightarrow \mathbb{N}, C^n(x_1, \dots, x_n) = 0$
- **Projections** $P_i^n : \mathbb{N}^n \rightarrow \mathbb{N}, P_i^n(x_1, \dots, x_n) = x_i$
- **Successor function** $S : \mathbb{N} \rightarrow \mathbb{N}, S(x) = x + 1$

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.

Closure Operations: Composition

- Composition**

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

$$f(\vec{x}) = h(g_1(\vec{x}), \dots, g_n(\vec{x}))$$

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

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

Closure Operations: Primitive Recursion

- Primitive recursion**

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

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

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

Definition 1. A function is **primitive recursive (p.r.)** if it can be constructed from the basic functions by applying composition and primitive recursion.

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

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

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

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

Example: Multiplication and Addition

To get multiplication we use another recursion:

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

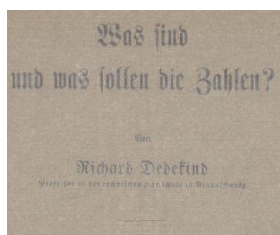
Here we use addition, which can in turn be defined by yet another recursion:

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

Since S is a basic function we have a proof that factorial are primitive recursive.

R. Dedekind

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



It is remarkable that a precise axiomatic description of the natural numbers involves computational ideas. In a sense, Dedekind reduces arithmetic to logic (the control structure of the computation) and the successor function.

A PR Programming Language

A while ago we promised a programming language, but all we have so far is a collection of functions.

So we need a programming language whose programs describe exactly the primitive recursive functions. This is rather too easy, but it's a good exercise in distinguishing between programs and their semantics.

The "programs" are really just terms constructed from some simple built-ins (for constant zero, projection and successor) plus constructors for composition and recursion. A pleasant feature of these terms is the complete absence of variables.

That's it. No control structures, no classes, no pointers, no arrays.

The Language PRec

Each term in our language has a fixed *arity*, corresponding to the number of arguments of the function it denotes. The built-in terms are

- C^n of arity n for each $n \geq 0$.
- P_i^n of arity n for each $n \geq i \geq 1$.
- S of arity 1.

Compound terms are constructed as follows:

- Given n terms τ_i of arity m and one term σ of arity n $\text{Comp}[\sigma, \tau_1, \dots, \tau_n]$ is a new term of arity m .
- Given terms σ of arity $n+2$ and τ of arity n $\text{PRec}[\sigma, \tau]$ is a new term of arity $n+1$.

Done.

Wouldn't it be nice if Java had a one-page description like this?

The Meaning of a Program

We have to explain what these "programs" mean.

Each well-formed term τ of arity n is associated with its meaning $\llbracket \tau \rrbracket$, a function:

$$\llbracket \tau \rrbracket : \mathbb{N}^n \rightarrow \mathbb{N}.$$

The definition is by induction on the build-up of τ , first the basic terms and then the compound ones.

Note that this is very different from Turing machines: there is no general way to do induction on the "structure of a Turing machine" (or on the number of states, see the Busy Beaver Problem). In this sense, primitive recursive functions are much more attractive.

Semantics

- *constants zero*

$$\llbracket C^n \rrbracket(x_1, \dots, x_n) = 0$$

- *projections*

$$\llbracket P_i^n \rrbracket(x_1, \dots, x_n) = x_i$$

- *successor function*

$$\llbracket S \rrbracket(x) = x + 1$$

- *Composition*

$$\llbracket \text{Comp}[\sigma, \tau_1, \dots, \tau_n] \rrbracket(\vec{x}) = \llbracket \sigma \rrbracket(\llbracket \tau_1 \rrbracket(\vec{x}), \dots, \llbracket \tau_n \rrbracket(\vec{x}))$$

- *Primitive recursion*

$$\llbracket \text{PRec}[\sigma, \tau] \rrbracket(\vec{x}) = \text{Prec}(\llbracket \sigma \rrbracket, \llbracket \tau \rrbracket)(\vec{x})$$

It follows immediately that $\llbracket \tau \rrbracket$ is always a primitive recursive function, and that all primitive recursive functions are obtained this way.

So Why Bother At All?

First off, our little language allows for very concise descriptions primitive recursive functions.

For example, factorial corresponds to

$$\text{Prec}[\text{Prec}[\text{Prec}[S \circ P_2^3, P_1^1] \circ (P_2^3, P_3^3), C^1] \circ (S \circ P_1^2, P_2^2), S \circ C^0]$$

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

This is hard to read for humans, but it shows very clearly that factorial is three applications of Prec away from the successor function (at least it is not clear how to do this with just two).

So we get a kind of complexity hierarchy.

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.

$$\text{add} = \text{Prec}[S \circ P_2^3, P_1^1]$$

$$\text{mult} = \text{Prec}[\text{add} \circ (P_2^3, P_3^3), C^1]$$

$$\text{pred} = \text{Prec}[P_1^2, C^0]$$

$$\text{sub}' = \text{Prec}[\text{pred} \circ P_2^3, P_1^1]$$

$$\text{sub} = \text{sub}' \circ (P_2^2, P_1^2)$$

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 1. Show that all these functions behave as expected.

Evaluation

Another reason our little language is useful is that we can clearly explain what we mean by an *evaluation operator*. We need an operator eval such that for any well-formed term τ of arity n and input $\vec{x} = x_1, \dots, x_n \in \mathbb{N}$ we have

$$\text{eval}(\tau, \vec{x}) = \llbracket \tau \rrbracket(\vec{x})$$

Note that the term τ is just a string, a word in some suitable alphabet.

Exercise 2. Write a compiler that given any string τ checks whether it is a well-formed expression denoting a primitive recursive function.

Exercise 3. Write an interpreter for primitive recursive functions (i.e., implement eval).

Primitive Recursion Functions and Logic

Enhancements

Another reason the programming language perspective helps a little is that it makes easy to identify missing features. For example, 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.?

And how about more complicated operations such as the GCD or the function that enumerates prime numbers?

Extending the Language: Definition by Cases

In order to get some definition-by-cases construct we have to be able to express relations (such $x < y$) in terms of primitive recursive functions, and we need to somehow express the control structure as a primitive recursive function.

Definition 2. Let $g, h : \mathbb{N}^n \rightarrow \mathbb{N}$ and $R \subseteq \mathbb{N}^n$.

Define $f = \text{DC}[g, h, R]$ by

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

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.

Note that we need express the relation R as a function; more on 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

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

so that $\text{sign}(0) = 0$ and $\text{sign}(x) = 1$ for all $x \geq 1$. Sign is primitive recursive: $\text{Prec}[S \circ C^2, C^0]$

Similarly the *inverted sign* function is primitive recursive:

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

Equality and Order

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

$$E = \overline{\text{sign}} \circ \text{add} \circ (\text{sub} \circ (P_1^2, P_2^2), \text{sub} \circ (P_2^2, P_1^2))$$

Or sloppy but more intelligible:

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

Then $E(x, y) = 1$ iff $x = y$, and 0 otherwise. Hence we can express equality as a primitive recursive function.

Even better, we can get other order relations such as \leq , $<$, \geq and $>$.

So, the fact that our language lacks relations is not really a problem; we can express them as functions.

Relations

Translating relations into functions is an interesting idea and deserves a general definition.

Definition 3. Suppose $R \subseteq \mathbb{N}^n$ is some relation. The *characteristic function* of R is defined by $\text{char}_R : \mathbb{N}^n \rightarrow \{0, 1\}$

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

Definition 4. A relation is *primitive recursive* if its characteristic function is primitive recursive.

The same method works for any notion of computable function: given a class of functions (Turing computable, p.r., polynomial time, whatever) one can deal with relations by considering those whose characteristic functions are in the class.

Closure Properties

Proposition 1. *The primitive recursive relations are closed under intersection, union and complement.*

Proof.

$$\begin{aligned}\text{char}_{R \cap S} &= \text{mult} \circ (\text{char}_R, \text{char}_S) \\ \text{char}_{R \cup S} &= \text{sign} \circ \text{add} \circ (\text{char}_R, \text{char}_S) \\ \text{char}_{\overline{R}} &= \text{sub} \circ (S \circ C^n, \text{char}_R)\end{aligned}$$

□

The proof is slightly different from the argument for (Turing) decidable relations but it's really the same idea.

Exercise 4. *Show that every finite set is primitive recursive.*

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.

Incidentally, primitive recursive functions were used extensively by K. Gödel in his incompleteness proof.

DC is Admissible

Proposition 2. *If g, h, R are primitive recursive, then $f = \text{DC}[g, h, R]$ is also primitive recursive.*

Proof.

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

Less cryptically

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

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

Bounded Sum

There are many other admissible operations.

Proposition 3. *Let $g : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ be primitive recursive, and define $f(x, \vec{y}) = \sum_{z < x} g(z, \vec{y})$.*

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

Proof.

$$\text{Prec}[\text{add} \circ (g \circ (P_2^{n+3}, \dots, P_{n+2}^{n+2}), C^n)]$$

□

Exercise 5. *Show that $f(x, \vec{y}) = \sum_{z < h(x)} g(z, \vec{y})$ is primitive recursive when h is primitive recursive and strictly monotonic.*

Bounded Search

A particularly important algorithmic technique is search over some finite domain. We can model search in the realm of p.r. functions as follows.

Definition 5. *Bounded Search*

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

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

Proposition 4. *If g is primitive recursive then so is $\text{BS}[g]$.*

This would usually be expressed as "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 but it can extend to a primitive recursive function of x and \vec{y} .

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

Dire Warning:

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

$$\min(z \mid g(z, \vec{y}) = 0)$$

is not an admissible operation; not even when there is a suitable z for each \vec{y} .

PR and Basic Number Theory

Claim 1. *The divisibility relation $\text{div}(x, y)$ is primitive recursive.*

Note that

$$\text{div}(x, y) \iff \exists z, 1 \leq z \leq y (x * z = y)$$

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 primitive recursive. But then

$$\text{sign} \left(\sum_{z \leq y} M(z, x, y) \right)$$

is the primitive recursive characteristic function of div .

Primality

Claim 2. *The primality relation is primitive recursive.*

Intuitively, this is true since x is prime iff

$$1 < x \text{ and } \forall z < x (\text{div}(z, x) \rightarrow z = 1).$$

Claim 3. *The next prime function $f(x) = \min(z > x \mid z \text{ prime})$ is p.r..*

This follows from the fact that bounded search again suffices:

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

Note: This bounding argument requires number theory (that's a white lie).

Claim 4. *The function $n \mapsto p_n$ where p_n is the n th prime is primitive recursive.*

Yet More Logic

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

Definition 6. *Bounded quantifiers*

$$P_{\forall}(x, \vec{y}) \iff \forall z < x P(z, \vec{y}) \quad \text{and} \quad P_{\exists}(x, \vec{y}) \iff \exists z < x P(z, \vec{y}).$$

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

Informally,

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

and likewise for P_{\exists} .

Note that the number of conjunctions on the right depends on x , so admissibility does not simply follow from closure under intersection.

Bounded Quantification

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

Generalizes to $\exists z < h(x, \vec{y}) P(z, y)$ and $\forall z < h(x, \vec{y}) P(z, y)$.

Proposition 5. *Primitive recursive relations are closed under bounded quantification.*

Proof.

$$\begin{aligned} \text{char}_{P_{\forall}}(x, \vec{y}) &= \prod_{z < x} \text{char}_P(z, \vec{y}) \\ \text{char}_{P_{\exists}}(x, \vec{y}) &= \text{sign} \left(\sum_{z < x} \text{char}_P(z, \vec{y}) \right) \end{aligned}$$

□

Again, the p.r. bound is needed, we cannot deal with $\exists z P(z, y)$ and $\forall z P(z, y)$.

Exercises

Exercise 6. *Give a proof that primitive recursive functions are closed under definition by multiple cases.*

Claim 5. *Show that the function $n \mapsto p_n$ where p_n is the n th prime is indeed primitive recursive.*

Data Structures

The Lack of Data structures

Data structures of all kinds are also glaringly absent from our language: we have no lists, trees, graphs, hash tables and so on, only integers.

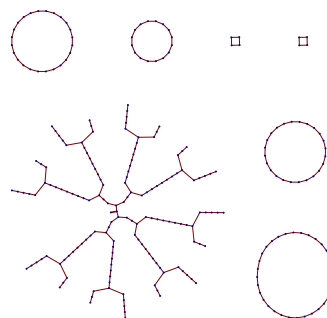
As it turns out, all these discrete structures can be obtained from just integers if we are able to express *sequences* a_1, a_2, \dots, a_n of numbers as a single number $\langle a_0, a_1, a_2, \dots, a_{n-1} \rangle$.

This is not meant as a practical programming idea, it is purely conceptual: integers already suffice in principle.

As a consequence, it makes sense to use families such as primitive recursive functions as a measure of complexity, even for algorithms that do not operate on integers. An algorithm that fails to have primitive recursive complexity is most likely utterly useless in practice.

Example: Graphs

For example, suppose we wish to implement graphs (below is the phase space of an elementary cellular automaton).



Graphs to Integers

We may safely assume that the vertices are integers $1, \dots, n$. Then a single edge is a pair (sequence of length 2) of integers, and all the edges can be expressed as a list of such pairs:

$$\langle \langle a_1, b_1 \rangle, \dots, \langle a_m, b_m \rangle \rangle$$

If coding is done the right way, we can still read off all the relevant information from this representation.

E.g., there should be a primitive recursive function

$$\text{edgeq}(G, x, y)$$

that returns 1 if (x, y) is an edge in graph G (all variables are integers), and 0 otherwise.

Coding Sequences

We write \mathbb{N}^* for all sequences of natural numbers.

We want an easy-to-compute, injective coding function that converts sequences of numbers into a single number

$$\langle \cdot \rangle : \mathbb{N}^* \rightarrow \mathbb{N}$$

and a decoding function that extracts the components of a coded sequence

$$\text{dec} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

so that

$$a_i = \text{dec}(\langle a_0, a_1, a_2, \dots, a_{n-1} \rangle, i)$$

Warm-Up: Pairing

There is a primitive recursive bijection $\pi : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, a so-called *pairing function*. For example, the following works:

$$\pi(x, y) = \frac{(x + y + 1)(x + y)}{2} + x$$

0	1	3	6	10	15	21
2	4	7	11	16	22	29
5	8	12	17	23	30	38
9	13	18	24	31	39	48
14	19	25	32	40	49	59
20	26	33	41	50	60	71
27	34	42	51	61	72	84

The table shows the values of π for $0 \leq x, y \leq 6$.

Note the counter-diagonal pattern of enumeration.

Unpairing

For a pairing function to be useful, we also need corresponding unpairing functions:

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

In our case, these functions have to primitive recursive.

Exercise 7. Show that the unpairing functions π_1 and π_2 are primitive recursive.

Exercise 8. Find one other primitive recursive pairing function with primitive recursive unpairing functions.

Sequence Numbers

Write p_n for the n th prime, so that the function $n \mapsto p_n$ is p.r.

Definition 7. Define a coding function $\langle \cdot \rangle$ by

$$\langle \vec{a} \rangle = p_1^{a_1+1} p_2^{a_2+1} \dots p_n^{a_n+1}$$

Also let

$$\text{Seq} = \{ \langle \vec{a} \rangle \mid \vec{a} \in \mathbb{N}^* \} \subseteq \mathbb{N}$$

be the set of all so-called **sequence numbers**.

Technically, $\langle \cdot \rangle$ cannot be primitive recursive since the domain \mathbb{N}^* is not allowed. However, for every fixed n , there is a p.r. function $f(a_1, \dots, a_n) = \langle \vec{a} \rangle$ for every fixed n .

Lemma 1. Seq is primitive recursive.

Decoding

Decoding comes down to computing prime factorizations:

$$a = p_1^{e_1} p_2^{e_2} \dots p_n^{e_n}$$

Exercise 9. Show that the function

$$\nu_p(x) = \max\{e \mid p^e \text{ divides } x\}$$

is primitive recursive.

Exercise 10. Show that the decoding function dec is primitive recursive.

Recording History

One 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:

$$f(x, \vec{y}) = z \iff \exists s \in \text{Seq} \left(\text{lh}(s) = x^+ \wedge (s)_0 = g(\vec{y}) \wedge \forall 1 < i < x ((s)_{i+} = h(i, (s)_i, \vec{y})) \wedge (s)_x = z \right)$$

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 :

$$\vec{f}(x, \vec{y}) = \langle f(0, \vec{y}), f(1, \vec{y}), \dots, f(x, \vec{y}) \rangle$$

Lemma 2. f is primitive recursive iff \vec{f} is primitive recursive.

Course of Value Recursion

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.

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

Lemma 3. If g and H are primitive recursive then f is also primitive recursive.

Exercise 11. Prove the last two lemmata. You may safely assume that standard sequence operations such as *append* are primitive recursive.

Digression: Gödel's β Function

There is more elegant 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 4. Gödel

There exists a primitive recursive function $\text{dec} : \mathbb{N}^2 \rightarrow \mathbb{N}$ such that

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

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

Proof. Set

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

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, \dots, a_{n-1} arbitrary and set

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

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

But then

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

□

Sequence Numbers

Definition 8. Define a coding function $\langle \cdot \rangle$ by

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

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

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

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

Exercise 12. 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.

Exercises

Exercise 13. Prove that all these functions are indeed primitive recursive, either for the prime-power coding function or for Gödel's method.

Exercise 14. Explain how to implement search in binary search trees as a primitive recursive operation.

Exercise 15. Come up with yet another coding function based on repeated application of a pairing function (something along the lines of $\langle a, b, c \rangle = \pi(a, \pi(b, c))$), but make sure your method really works).

Turing Machines versus Primitive Recursive Functions

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 6. *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 $T: \mathbb{Z} \rightarrow \Sigma$, $i \in \mathbb{Z}$ 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.

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

which list can be coded by a single integer.

t Steps

Hence we can encode a whole sequence of IDs

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

again by a single integer and check in a p.r. way that $C_i \vdash_M^1 C_{i+1}$.

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 16. *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 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.

Exactly How Much is Missing?

The gap between Turing machines and p.r. functions seems to stem from the fact that though each step in a Turing machine is easily p.r., there is no obvious p.r. bound on the total number of steps.

So, can we concoct a Turing computable function that fails to be primitive recursive?

The cheap answer is that p.r. functions are always total whereas Turing computable functions in general are not.

A more interesting challenge is:

► Construct a total function that is Turing computable but not p.r..

One way to do this is to make sure the function grows faster than any primitive recursive one, again by exploiting the inductive structure of these functions.

Ackermann's Function (1928)

The Ackermann function $A: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ is binary and defined by

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

where x^+ is shorthand for $x + 1$.

Note the odious double recursion – on the surface, this looks more complicated than primitive recursion.

Ackermann is Computable

Here is a bit of C code that implements the Ackermann function (assuming that we have infinite precision integers).

```
int acker(int x, int y)
{
    return x ? ( acker( x-1, y ? acker( x, y-1 ) : 1 ) ) : y+1 ;
}
```

All the work of organizing the nested recursion is handled by the compiler and the execution stack.

Of course, doing this on a Turing machine is a bit more challenging, but it can be done.

Family Perspective

It is useful to think of Ackermann's function as a family of unary functions $(A_x)_{x \geq 0}$ where $A_x(y) = A(x, y)$.

The critical part of the definition then looks like so:

$$A_{x+1}(y) = \begin{cases} A_x(1) & \text{if } y = 0, \\ A_x(A_{x+1}(y-1)) & \text{otherwise.} \end{cases}$$

From this it follows easily by induction that

Claim. Each function A_x is primitive recursive.

The Mystery of $A(4,1)$

$$A(0, y) = y^+$$

$$A(1, y) = y^{++}$$

$$A(2, y) = 2y + 3$$

$$A(3, y) = 2^{y+3} - 3$$

$$A(4, y) \approx 2^{2^{2^{\dots^2}}}$$

$$A(5, y) \approx \text{super-super exponentiation}$$

$$A(6, y) \approx \text{an unspeakable horror}$$

$$A(7, y) \approx ???$$

The first 4 levels of the Ackermann hierarchy are easy to understand, A_4 starts causing problems, and beyond A_5 things quickly spin out of control.

Ackermann and Union/Find

From the table, one might suspect that the Ackermann is a purely theoretical construct whose only purpose in life it is to refute the conjecture that computable is the same as primitive recursive.

Surprisingly, the Ackermann function pops up in the analysis of the well-known Union/Find algorithm (with ranking and path-compression), a method to compute dynamic equivalence relations.

The running time of Union/Find differs from linear only by a minuscule amount, which is something like the inverse of the Ackermann function.

But in general anything beyond level 3.5 of the Ackermann hierarchy is irrelevant for practical computation.

Exercise 17. Read an algorithms text that analyzes the run time of the Union/Find method.

Ackermann is Total

Note that it is not entirely clear that $A(x, y)$ is well-defined for all x and y , see the notes on well-orders.

One possible proof is by induction on x , and subinduction on y .

A more elegant way is to use induction on $\mathbb{N} \times \mathbb{N}$ with respect to the lexicographic product order

$$(a, b) < (c, d) \iff (a < c) \vee (a = c \wedge b < d),$$

a well-ordering of order type ω^2 .

Exercise 18. Carry out the proof of totality.

Ackermann vs. PR

Theorem 1. The Ackermann function dominates every primitive recursive function in the sense that there is a k such that

$$f(\vec{x}) < A(k, \max \vec{x}).$$

Hence A is not primitive recursive.

Sketch of proof.

One can argue by induction on the buildup of f .

The atomic functions are easy to deal with.

The interesting part is to show that the property is preserved during an application of composition and of primitive recursion. Alas, the details are rather tedious.

□

Turing vs. Ackermann

Let's come back to the gap between primitive recursive and Turing computable functions.

How much do we have to add to primitive recursion to capture the Ackermann function?

Proposition 7. *There is a primitive recursive relation R such that*

$$A(a, b) = c \iff \exists t R(t, a, b, c)$$

Think of t as a complete trace of the computation of the Ackermann function on input a and b . For example, t could be a hash table that stores all the values that were computed during the call to $A(a, b)$ using memoizing.

Suppose someone gives you a hash table t and claims that it is the result of the call to $A(a, b)$ and produces output c . How can you check whether this is true?

Verifying Traces

The entries in t are determined by the following rules: given a and b

- $(a, b, c) \in t$ for some c
- $(0, y, z) \in t$ implies $z = y^+$
- $(x^+, 0, z) \in t$ implies $(x, 1, z) \in t$
- $(x^+, y^+, z) \in t$ implies $(x, u, z) \in t$ and $(x^+, y, u) \in t$ for some u .
- $(x, y, z) \in t$ and $(x, y, z') \in t$ implies $z = z'$.

Any table that satisfies these rules proves that indeed $A(a, b) = c$ (though it might have extraneous entries).

This should look familiar: we have a decision algorithm that tests whether an alleged computation of A is in fact correct.

In fact, it is not hard to show that the decision algorithm is primitive recursive.

Unbounded Search

So to compute A we only need to add search: systematically check all possible tables until the right one pops up (it must since A is total). The problem is that this search is no longer primitive recursive.

More precisely, let

$$\begin{aligned} \text{acker}(t, x, y) &\iff t \text{ is a correct trace for } x, y \\ \text{lookup}(t, x, y) = z &\iff (x, y, z) \text{ in } t \end{aligned}$$

Then

$$A(x, y) = \text{lookup}(\min(t \mid \text{acker}(t, x, y)), x, y)$$

This is all primitive recursive except for the unbounded search in \min .

General Recursive Functions

Closing the Gap

Since the Ackermann function is nearly primitive recursive it is tempting to extend primitive recursive a bit by adding a \min operator.

From the previous discussion, primitive recursive plus \min operator is powerful enough to produce the Ackermann function – of course, the real question is: what is relationship between the extended class and Turing-computable functions in general?

As we will see, we obtain the same class of functions.

(General) Recursive Functions

Definition 9. *The class of general recursive functions is defined like the primitive recursive functions, but with one additional operation: unbounded search.*

$$f(\vec{x}) = \min(z \mid g(z, \vec{x}) = 0)$$

Here g is required to be total.

Thus, $f(\vec{x}) = 3$ means $g(3, \vec{x}) = 0$, but $g(2, \vec{x}), g(1, \vec{x}), g(0, \vec{x}) > 0$.

Note: Even though we insist that g is total, f will in general be a partial function: for some \vec{y} there may well be no z .

But one cannot allow for g itself to be partial; there are examples that show that this would violate computability.

Terminology

Other names in the literature: *partial recursive functions* or *μ -recursive functions*.

The notion μ -recursive function comes from the fact that in the older literature one usually finds

$$f = \mu g$$

rather than a reference to \min .

The notion of "general recursive function" was used because primitive recursive functions were originally referred to as "recursive functions".

To round off the confusion, some authors mean total computable functions when they say "recursive function", and use "partial recursive function" for the general case.

We will use the latter convention (well, most of the time anyways).

Decidability

Definition 10. A relation $R \subseteq \mathbb{N}^n$ is **decidable** if its characteristic function char_R is recursive.

From the old proof for p.r. relations we get:

Lemma 5. The decidable subsets of \mathbb{N} are closed under union, intersection and complement.

They are also closed under bounded quantification.

But they are not closed under unbounded quantification.

Surprisingly (to Hilbert) there are lots of undecidable problems in mathematics.

Regular Search

Let us call a function g *regular* if

$$\forall \vec{y} \exists z g(z, \vec{y}) = 0.$$

Hence, if we define f from g by minimization, the resulting function will be total.

Define the *regular search computable (RSC)* functions to be the class obtained from:

- The basic functions zero, projections, addition and multiplication, and
- closed under composition and regular search.

What class of functions do we obtain this way?

RSC is somewhat similar to primitive recursive, but, at least at first glance, it is hard to compare the relative power of primitive recursion versus regular search; the operations are too different.

Sequence Numbers and RSC

With a little bit of effort one can show that the whole coding and decoding machinery required to obtain sequence numbers is all RSC. But then one can show that primitive recursion is admissible with respect to RSC.

To see this, suppose g and h are RSC and let $f = \text{Prec}[h, g]$. Recall our characterization of primitive recursion in terms of sequence numbers:

$$f(x, \vec{y}) = z \iff \exists s \in \text{Seq} \left(\text{lh}(s) = x^+ \wedge (s)_0 = g(\vec{y}) \wedge \forall 1 < i < x ((s)_{i+} = h(i, (s)_i, \vec{y})) \wedge (s)_x = z \right)$$

Now let $H(s, x, \vec{y})$ be the function that is 0 if the matrix of the formula on the right holds, and 1 otherwise. Then

$$f(x, \vec{y}) = \min(s \mid H(s, x, \vec{y}) = 0)$$

shows that f is RSC.

Church-Turing Thesis

Let $f : \mathbb{N}^n \rightarrow \mathbb{N}$ be a function. Then the following are equivalent:

- f is intuitively computable,
- f is general recursive,
- f is Turing computable.

Of course, there is a proper theorem hiding here: general recursive is provably the same as Turing computable. But, we cannot hope to formally establish the equivalence with intuitive notions of computability.

There are many other equivalent characterizations which all add to the weight of this thesis.

Also note that Gödel was completely convinced by Turing's analysis (and gave him full credit, claiming none for himself and Church).

The Interesting Part

Kleene established a strong normal form result that shows that all computable functions can be represented in a certain uniform way.

Theorem 2. *Kleene Normal Form*

There is a p.r. relation trace and a p.r. function h such that Turing machine M_e on input x halts on output y if, and only if,

$$y = h(\min(t \mid \text{trace}(t, e, x))).$$

Here t is a complete trace of the computation of M_e on input x , let's say (the code number of) a list of all the IDs occurring in the computation. The trace predicate checks whether t is really a correct trace for machine M_e on input x .

Note the h : it extracts the output of M from the trace t . One can show that some device such as h is required.

Traces

Again: we cannot in general get a p.r. bound on the trace t .

The situation is this:

$$\mathcal{T} = \{ (t, e, x) \in \mathbb{N}^3 \mid \text{trace}(t, e, x) \} \subseteq \mathbb{N}^3$$

is p.r., and in fact it is very easy to decide membership in \mathcal{T} . But

$$\{ (e, x) \in \mathbb{N}^2 \mid \exists t ((t, e, x) \in \mathcal{T}) \} \subseteq \mathbb{N}^2$$

is undecidable. Even

$$\{ e \in \mathbb{N} \mid \exists t ((t, e, e) \in \mathcal{T}) \} \subseteq \mathbb{N}$$

is undecidable.

Kleene's T Predicate

Similar results hold for functions with several arguments:

The relation

$$T = \{ (t, e, \vec{x}) \in \mathbb{N}^{k+2} \mid \text{trace}(t, e, \vec{x}) \} \subseteq \mathbb{N}^{k+2}$$

is traditionally called Kleene's T -predicate.

By performing an unbounded search over t we can compute any k -ary function: we only need to select the proper value for e .

So e plays the same role as a program given to an interpreter.

Enumerations

By the Kleene Normal Form theorem we can give an enumeration of all partial recursive functions. One often writes

$$\{e\}$$

for the e th function. The number e is an *index* for the function $\{e\}$.

Actually, one should write something like $\{e\}^k$ to indicate that the function has arity k , but we will mostly ignore such details.

Since these functions are partial one has to be a bit careful with equations. One often writes

$$\{e\}(x) \simeq y$$

to indicate that $\{e\}$ with input x returns output y after finitely many steps.

Stages of a Computation

One writes

$$\{e\}_s(x) \simeq y$$

to indicate that $\{e\}$ with input x returns output y after at most s steps. Hence

$$\{e\}(x) \simeq y \iff \exists s (\{e\}_s(x) \simeq y)$$

But note that the bound s cannot be computed: there is no total recursive function f such that

$$\{e\}(x) \simeq y \iff \exists s < f(e, x) (\{e\}_s(x) \simeq y)$$

Convergence and Divergence

When dealing with partial recursive functions it is convenient to have some shorthand notation for convergence:

$$\{e\}(x) \downarrow$$

means that the computation halts after finitely many steps. Likewise,

$$\{e\}(x) \uparrow$$

means that the computation does not produce any output.

Note that the first property is semi-decidable, but the latter is not (it is, of course, co-semi-decidable).

Partial versus Total

It might be tempting to think of partial recursive functions as total recursive functions restricted to some smaller domain of definition. Tempting, but very wrong.

Proposition 8. *There is a partial recursive function g that is not the restriction of any total recursive function.*

Proof.

Let

$$g(x) = \begin{cases} \{e\}(e) + 1 & \text{if } \{e\}(e) \downarrow, \\ \uparrow & \text{otherwise.} \end{cases}$$

It is easy to see that g cannot be of the form $g = f \upharpoonright D$ for any total recursive function f .

□

Kleene's S - m - n Theorem

Once we have a fixed enumeration, one can compute with indices.

Here is a particularly useful, though admittedly quite unspectacular instance of such an index computation: we can fix some of the arguments of a computable function to obtain another computable function.

Theorem 3. *For every $m, n \geq 1$ there is a primitive recursive function S_n^m such that*

$$\{S_n^m(e, \vec{p})\}^n(\vec{x}) = \{e\}^{m+n}(\vec{p}, \vec{x}).$$

Another example of index computation: there is primitive recursive function f such that

$$\{f(e, e')\} = \{e\} \circ \{e'\}.$$

Given two programs, we can compute a new program that represents the composition of the given ones.

Summary

- There are several approaches towards defining computability.
- Different models may well turn out to be equivalent, e.g., Turing computable is the same as recursive.
- Primitive recursive functions are much stronger than actually computable functions, but fail to completely capture the notion of computability.
- Recursive functions encapsulate the idea of unbounded search.
- Ackermann and Busy Beaver explode at surprisingly low levels.
- Church's Thesis states that Turing computability precisely captures the intuitive notion of computability.
- Kleene's Normal Form theorem is an important tool when dealing with computable functions.