

# Primitive and General Recursive Functions

Klaus Sutner  
Carnegie Mellon University

Fall 2010

# Outline

- 1 Primitive Recursive Functions
- 2 Primitive Recursion and Logic
- 3 Register Machines versus Primitive Recursion
- 4 Other Models

## 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.

We will define 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 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

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

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

$g$  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:

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

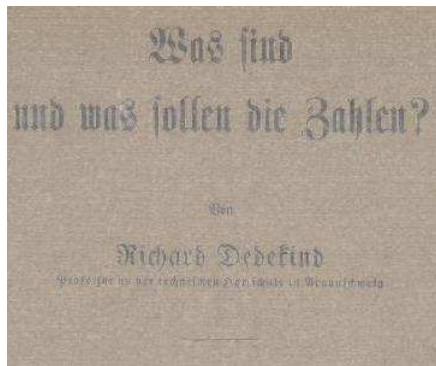
Here we use addition, which can in turn be defined by yet another recursion:

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

Since  $S$  is a basic function we have a proof that multiplication is 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?"



## 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

*Show that all these functions behave as expected.*

- Primitive Recursive Functions

- ② Primitive Recursion and Logic

- Register Machines versus Primitive Recursion

- Other Models

## Enhancements

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?

## Definition by Cases

### Definition

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

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

## Relations

As before we can use the **characteristic function** of a relation  $R$

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

to translate relations into functions.

### Definition

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

The same method works for any notion of computable function: given a class of functions (RM-computable, p.r., polynomial time, whatever).

## Closure Properties

### Proposition

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

*Proof.*

$$\text{char}_{R \cap S} = \text{mult} \circ (\text{char}_R, \text{char}_S)$$

$$\text{char}_{R \cup S} = \text{sign} \circ \text{add} \circ (\text{char}_R, \text{char}_S)$$

$$\text{char}_{\mathbb{N} - R} = \text{sub} \circ (S \circ C^n, \text{char}_R)$$

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

### Exercise

*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

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

*Proof.*

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

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

### Proposition

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.*

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

### Exercise

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.

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

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

## Closure

One can show that bounded search adds nothing to the class of p.r. functions.

### Proposition

*If  $g$  is primitive recursive then so is  $BS[g]$ .*

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

*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  $x \cdot z = y$  is p.r. But then

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

is the p.r. characteristic function of  $\text{div}$ .

## Primality

### Claim

*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

*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:

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

This bounding argument requires number theory (a white lie).

## Enumerating Primes

### Claim

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

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

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

## 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

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

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  $P_{\exists}$ .

## 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

*Primitive recursive relations are closed under bounded quantification.*

*Proof.*

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

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

□

## Exercises

### Exercise

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

### Exercise

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

- Primitive Recursive Functions
- Primitive Recursion and Logic
- ③ Register Machines versus Primitive Recursion
  - Other Models

## RM vs. PR

**Burning Question:** How does the computational strength of register 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 RM.

This comes down to building a RM 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 RM.

## Opposite Direction?

- The cheap answer is to point out that some RM-computable functions are not total, so they cannot be p.r..
- True, but utterly boring. Here are the right questions:
- How much of a RM computation is primitive recursive?
- Is there a total RM-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 “RM  $M$  moves from configuration  $C$  to configuration  $C'$  in  $t$  steps” is primitive recursive. relation).

But when we try to deal with “RM  $M$  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 RM-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 register machine. The  $t$ -step relation*

$$C \left| \begin{array}{c} t \\ M \end{array} \right. C'$$

*is primitive recursive, uniformly in  $t$  and  $M$ .*

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

Since configurations are of the form

$$(p, (x_1, \dots, x_n))$$

where  $p$  and  $x_i$  are natural numbers this is a straightforward application of sequence numbers.

## $t$ Steps

Likewise we can encode a whole sequence of configurations

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

again by a single integer.

And we can check in a p.r. way that  $C \stackrel{t}{\underset{M}{\mid}} C'$ .

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

$$C \stackrel{M}{\vdash} C'$$

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.

## Exactly How Much is Missing?

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

The cheap answer is that p.r. functions are always total whereas register computable functions in general are not. A much more interesting challenge is:

- ▶ Construct a total function that is RM-computable but not primitive recursive.

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 the these functions.

## Ackermann's Function (1928)

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

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

$$A(x^+, 0) = A(x, 1)$$

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

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 register 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 $A(6, \cdot)$

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

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

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

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

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

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

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

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

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

One might think that the only purpose of the Ackermann function is to refute the claim that computable is the same as p.r. Surprisingly, the function pops up in the analysis of the Union/Find algorithm (with ranking and path compression).

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

*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 we 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  $\omega^2$ .

### Exercise

*Carry out the proof of totality.*

## Ackermann vs. PR

### Theorem

*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.



## Register vs. Ackermann

Let's come back to the gap between p.r. and RM-computable.

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

### Proposition

*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.

## Verifying Traces

The entries in  $t$  are determined by the following rules:

- $(a, b, c) \in t$  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).

So 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  $\min$ .

## Like Kleene

This should look eminently familiar: Kleene's  $T$  predicate does the same for all computable functions.

$$P_e(x) \downarrow \iff \exists t T(e, x, t)$$

$$P_e(x) = U(\min(t \mid T(e, x, t)))$$

If we choose  $e$  to be an index for the Ackermann function this will produce exactly the result from the previous slide.

- Primitive Recursive Functions
  - Primitive Recursion and Logic
  - Register Machines versus Primitive Recursion
- 4 Other Models

## Primitive Recursive is Trivial

In the context of general computability theory “primitive recursive” translates roughly into “easily computable.”

In some theories it is convenient to assume that all primitive recursive functions and relations are simply given as atomic objects, there is no need to dig any deeper.

Of course, this has nothing to do with practical computability. Even just a handful of nested primitive recursions can well mean that the computation is not feasible.

## The Framework

But primitive recursive functions provide a nice framework for other models of computation.

- Define a class  $\mathcal{C}$  of possible configurations and code them up as natural numbers, so  $\mathcal{C} \subseteq \mathbb{N}$  is primitive recursive.
- Define a one-step relation on  $\mathcal{C}$  that is primitive recursive.
- Define some reasonable input/output conventions.

For example, Turing machines, random access machines, parallel random access machines all fall into this framework.

Other approaches such as programming languages, equational calculi or more abstract models such as Church's lambda calculus also fit into this framework, though not quite as easily.

## Church's Thesis

**Claim:** As long as this is done in a reasonable way, we always get the same notion of computability.

Incidentally, Gödel was hesitant to agree to Church's Thesis; he did change his mind once he saw Turing's 1936 paper.

## Summary

- There are several approaches towards defining computability.
- Different models may well turn out to be equivalent, e.g., register machine computable is the same as general 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.
- The Ackermann function explodes at surprisingly low levels.
- Church's Thesis states that various formal notions of computability precisely capture the intuitive notion of computability.