This version was compiled on November 27, 2002

Forms of recursion and induction

In these notes we look at the related techniques of recursion and induction. You will have seen already some of the simpler kinds of induction (over $\mathbb{N}$) and you may have met some forms of recursion (such as primitive recursion) but you may not be aware of this name. Here we will go beyond these rather simple particular cases, and look at some more intricate aspects.

Roughly speaking (at least for what we do here) recursion is a way of specifying a function by saying how each of its values is related to 'earlier' values. You may not be familiar with the word 'specifying' for it is often incorrectly replaced by 'defining'. The meaning and the difference between these two notions will be explained later.

The related technique of induction is more familiar. This is a method of proof used to verify certain facts about the functions produced by a specification. In general, for each form of recursion (which produces functions) there is an associated form of induction (which verifies properties of the functions).

In short, recursion is a way of producing functions, whereas induction is a way of proving properties (of functions). The aim of these notes is to make these notions clearer in a context more general than you have seen before. We will only begin to scratch the surface of the subject, and we will not go to its extremes. Nevertheless we will see some quite wild examples of the technique.

One of the long term aims of this part of mathematics is to explain why this function is more complicated than that function, and develop methods of measuring the difference. In turns out that recursion, in various forms, has a lot to say about this.

In these notes we concentrate at recursion and induction over $\mathbb{N}$, the natural numbers. However, you should be aware that the techniques are applicable in many other situation. For instance the natural numbers can be replaced by lists over any alphabet. In fact, the techniques can be used on any free algebra and, in particular, on syntactic constructions. We will some small instances of this in section 3, and much more in the notes on λ-calculi.

The ordinals are another domain where the techniques are useful. However, with these there is an extra twist that is needed. You should learn something about this sometime.

Contents

1 Introduction 2
  Exercises 8
2 Setting the scene 9
  Exercises 18
3 The development and complexity of arithmetic 22

Exercises 30
4 The primitive recursive functions 31
Exercises 38
5 Deep inside primitive recursion 41
Exercises 44
6 Way beyond primitive recursion 46
Exercises 53
7 Feasibililty and computability 55
Some references 59
Some solutions 61

1 Introduction

What are these two notions 'recursion' and 'induction' and why do they have 'forms'? There is no all embracing definition or characterization of these, but this doesn't matter for we don't need one here. You will have seen many examples of induction, and even though you might not know it, you will have seen some examples of recursion. On the whole these will be rather simple examples, and you might not realize that they are part of a quite large topic. In these notes we are going to look at a much broader class of examples, without ever going to the extremes. Before we do this let's try to see what the topic is about. We will start with what you know and extend these ideas a bit until we get to a suitable setting from which we can begin the development in earnest.

You will be familiar with induction over the natural numbers $\mathbb{N}$. Suppose $\phi(x)$ is assertion which depends on an arbitrary natural number x. (There may be other parameters involved, but we can think of those as fixed.) For any particular natural number m the instance $\phi(m)$ is either true or false. The quantified assertion

$$(\forall x : \mathbb{N})\phi(x)$$

says that $\phi(m)$ is true for every $m \in \mathbb{N}$.

(The syntax '$x : \mathbb{N}$' in the quantification is a reminder that x must range over $\mathbb{N}$. We will see more of this kind of typing restriction in λ-calculi. Here it doesn't matter too much if you don't fully understand how the construction is used.)

One way we might be able to prove the quantified assertion is by induction over $\mathbb{N}$. Thus we prove outright the base case

$$\phi(0)$$

and then prove the step case

$$(\forall x : \mathbb{N})[\phi(x) \Longrightarrow \phi(x + 1)]$$

which then generates all the particular cases

$$\phi(1), \phi(2), \phi(3), \ldots$$

in turn. Usually when we write out such a step we say we are proving

$$\phi(x) \Longrightarrow \phi(x+1)$$

on the understanding that this x is arbitrary. In other words, we tacitly assume that x is quantified.

This is the kindergarten version of induction beloved by school masters (except that they often take $\phi(1)$ as the base case). We will refer to it as the base/step form of induction.

Sometimes this b/s method doesn't work (because of the nature of ϕ). Sometimes we need to use the progressive form of induction. Thus we show

$$(\forall y < x)\phi(y) \Longrightarrow \phi(x)$$

for arbitrary x. Again this is enough to generate

$$\phi(0), \phi(1), \phi(2), \phi(3), \ldots$$

in turn. Notice that this form doesn't need a base case. However, in practice it is often convenient to verify $\phi(0)$ separately.

Occasionally other forms of induction over $\mathbb{N}$ are needed. For instance the two base cases $\phi(0), \phi(1)$ and a two-step case

$$\phi(x) \wedge \phi(x+1) \Longrightarrow \phi(x+2)$$

is enough to waltz through to $(\forall x : \mathbb{N})\phi(x)$. This is some where between the b/s version and the progressive version. (You might like to work out where you have seen instances of this form of induction.)

You will be familiar with much of this, but perhaps not set out in this concise fashion.

Induction is a very powerful proof method, and there are several observation that should be made. The above forms of induction are not the only ones that occur. You may not be familiar with many, or any, other forms, so it will do no harm to have a look at the details of some of them. In fact, it will do you a lot of good to have a look at some other forms. For instance, sometimes it is necessary to manipulate with two or more variables at once. As you can imagine, this could be problematic if the two variables start to interfere with each other and cause the various implications to become circular.

The natural numbers are not the only structure over which induction can be used. For instance, you may know something about the ordinals, for which more sophisticated forms of induction are needed. A common tool (in this corner of mathematics) is induction over syntax. You will have seen simple examples of this already, but perhaps this name is new. We will look at this here and elsewhere.

In short, induction is a common method of proof in many areas of mathematics, and you have seen some of this before. Here we are going to look at more complicated examples, and try to classify what these can do.

What about recursion? Again you will have seen some examples, but in this case almost trivial, never mind simple, examples.

Probably the most common introduction to recursion is the construction of the factorial function

$$fx = x!$$

by

$$f0 = 1 \qquad f(x+1) = (x+1) \times (fx)$$

(for $x \in \mathbb{N}$). This is not an explicit definition, but an algorithm for evaluating the function. To calculate $f7$ we must first calculate $f6$ and then multiply by 7. But to calculate $f6$ we must first calculate $f5$ and then multiply by 6; and so on until we bottom out at the explicit case $f0 = 1$.

This is the essence of recursion: repeatedly do something using the results of earlier calculations to perform later calculations. Clearly recursion has a lot in common with induction, but there is a significant difference. Here is a slogan you should remember.

Recursion is a method of Construction	Induction is a method of Proof

By the end of these notes you should have a better understanding of the differences and similarities between these notions.

As suggested earlier, there isn't an all embracing definition of 'recursion', and it isn't useful to try to cobble one together. However, it is useful to see some more sophisticated examples of the technique. You will then begin to see what it is about.

For our purposes recursion is a method of producing functions. Here we will almost always restrict out attention to functions over the natural numbers. However, as with induction, recursion can be used in other settings.

Let's first set up some examples, and then look at them in more detail.

1.1 EXAMPLES. (a) Consider the two 3-placed functions f (from natural numbers to natural numbers) given by

$$\begin{aligned} f(0,y,z) &= z \times y \\ f(x+1,y,z) &= z + f(x,y,z) \end{aligned} \qquad \begin{aligned} f(0,y,z) &= z \times y \\ f(x+1,y,z) &= z \times f(x,y,z) \end{aligned}$$

(for $x, y, z \in \mathbb{N}$).

(b) Consider the 1-placed function (over $\mathbb{N}$) given by

$$f0 = 0 \quad f1 = 1 \quad f(x+2) = f(x+1) + fx$$

(for $x \in \mathbb{N}$).

(c) Consider the 2-placed function f (over $\mathbb{N}$) given by

$$\begin{array}{rr} f(0,0)=1 & f(0,i+1)=0 \\ f(x+1,0)=1 & f(x+1,i+1)=f(x,i)+f(x,i+1) \end{array}$$

(for $x, i \in \mathbb{N}$). ■

Each of these is an example of a specification of a function by recursion. (Don't worry about about the meaning of 'specification', it will be explain later.) The common feature of each is that it is not an explicit definition, it is more like an algorithm which tells us how to evaluate the function.

Example 1.1(a) gives two different functions. The important thing is that they are both constructed in a similar way. Both constructions use the same *form* of recursion. Both are instances of

$$f(0,y,z)=g(y,z) \qquad f(x+1,y,z)=h(x,y,z,f(x,y,z))$$

where g and h are known functions. You may recognize this as primitive recursion. (For these particular examples the value $h(x,y,z,t)$ is independent of x and y.) It is easy to check that

$$f(x,y,z)=(x+y)z \qquad f(x,y,z)=yz^{x+1}$$

respectively. These identities are proved by a b/s induction over x. Notice that even though both functions are obtained by almost the same recursion, the slight change in the ingredients seems to produce a significant difference in the output. Later we will see a better way of viewing the outputs which highlights the similarities.

Example 1.1(b) produces the fibonacci function (which, the story is told, has something to do with breeding rabbits). It can be shown that there is a certain number τ such that

$$fx=\tau^x-\tau^{-x}$$

holds for all $x \in \mathbb{N}$. This is proved by a two-step induction over x once the value of τ is known. (Various new and old age nutters think that τ has some mystical significance).

The original specification gives us a way of evaluating f. This associated explicit definition gives a different way of evaluating f which, in some ways is more efficient. It turns out that τ is a real number greater than 1 (but not too large). In particular, τ^{-x} tends to 0 as x increases. Thus, except for the first few x, the values fx and τ^x differ by a small amount with τ^x the larger. But fx is a natural number, so that

$$fx=\lfloor\tau^x\rfloor$$

for all $x \geq 5$ (say). In other words, to calculate fx we can approximate τ^x and then round down to a natural number. (You thought it was only the finance industry who made constructive use of rounding down.)

It is instructive to see how this form of recursion can be converted into a simpler form. The original function

$$f : \mathbb{N} \longrightarrow \mathbb{N}$$

sends natural numbers to natural numbers. In its place we consider a function

$$F : \mathbb{N} \longrightarrow \mathbb{N}^2$$

which sends natural number to pairs of natural numbers and is given by

$$F0 = (0, 1) \qquad F(x + 1) = (r, l + r) \text{ where } Fx = (l, r)$$

(for $x \in \mathbb{N}$). This is another form of recursion which is very like primitive recursion, but strictly speaking is different.

Example 1.1(c) produces the binomial coefficients

$$f(x, i) = \binom{x}{i}$$

as generated by the Pascal triangle. (The recursion incorporates a little trick to convert the triangle into a square.) The crucial difference here is that it is a recursion over two variables x and i. It can be rephrased as a more standard recursion, but that needs a few more tricks.

These examples give some indication of what recursion does. It is a way of producing functions by describing an algorithm for evaluating the function. The crucial ingredient is that most values are calculated in terms of 'earlier' values which in turn are calculated in terms of even 'earlier' values, and so on. In short, the value of the function at an input is calculated by a regression through values at 'earlier' inputs.

In some cases an 'earlier' input need not be one of smaller value. Indeed, in some more general recursions the inputs may not have values that are comparable, so that 'earlier in value' doesn't make sense.

In some circumstances such a regression may carry on indefinitely, so more often than not a recursion contains a clauses which enables the algorithm to bottom out (that is, to terminate) at least for some inputs.

Some recursions can be convoluted and rather hard to analyse. We will look at some of the ways a recursion can be set up. By doing this we begin to develop a notion of the 'complexity' of a function. In the first instance this is merely a qualitative idea, but eventually various quantitative measures of complexity emerge.

Mostly we will look at recursion over $\mathbb{N}$. To start let's consider a $(1 + s)$-placed function f over $\mathbb{N}$. This will have a typical value

$$f(x_0, x_1, \ldots, x_s)$$

for inputs $x_0, x_1, \ldots, x_s$. To produce this by recursion we select one of the inputs as the recursion variable (over which the recursion takes place). Of course, if we select the wrong one then it may not be possible to use recursion. In fact, there may not be any sensible way of using recursion to produce the function. Let's suppose there is, so one of the inputs can act as the recursion variable. It is convenient to have this in a special position and distinguished from the other inputs. By a suitable shuffling of the inputs we can put the recursion variable in the left-hand position. (At other times a different place might be more useful). Thus the typical value is

$$f(x, y_1, \ldots, y_s)$$

where x is the recursion variable and $y_1, \ldots, y_s$ are the other inputs.

In general, in these notes, we will try to use 'x' as the recursion variable. There is nothing too significant with this, it just helps us to locate where the action is. Of course, there will be times when we have to break this convention.

The recursion variable x does one kind of job, and the other inputs $y_1, \ldots, y_s$ do another kind of job. We refer to these $y_1, \ldots, y_s$ as the parameters. Again there is nothing too significant with this, it is just a useful bit of terminology to help use differentiate between the different jobs.

Surprisingly, this linguistic trick leads to a cleaner view of the set up. Let us bundle together the parameters into an s-tuple

$$p = (y_1, \ldots, y_s)$$

and think of this as a single parameter. Thus rather than receiving $1 + s$ inputs, the function receives two, the recursion variable x and the parameter p, to produce

$$f(x, p)$$

its output. The form of the recursion (or at least the the kind we will start with) now looks something like

$$f(0, p) = \text{explicit in } p \qquad f(x + 1, p) = \text{obtained from 'earlier' values } f(\cdot, p)$$

where we need not unravel p (until we get into an actual evaluation). This simple condensation of the notation gives us a better and more general perspective of recursion.

- The recursion variable x should range over $\mathbb{N}$, at least for the recursions we are interested in.
- The parameter p need not be a tuple of natural numbers. It could be something quite different, such as another function.
- The output of the function need not be a natural number.

Thus we can think of a function

$$f : \mathbb{N} \times \mathbb{P} \longrightarrow \mathbb{T}$$

where $\mathbb{P}$ is some space of parameters and $\mathbb{T}$ is some space of target values.

By viewing a function in this way we get a better idea of the recursion involved. We are able to see more clearly the similarities and differences between recursions without being distracted by the irrelevant inner details.

To conclude this introduction let's go back to the slogan. As stated there, and as should be perfectly obvious, induction is a method of proof. However, you will often see the phrase 'defined by induction' especially when there is explicit syntax around. At best this is sloppy phraseology since nothing can be defined by induction. What is usually meant is that a very simple form of recursion, namely iteration, is being use to construct some gadgets (usually of a syntactic nature). If this was nothing more than sloppy language then it wouldn't be too bad. However, some people seem not to know the difference between construction and proof, and they can transmit their confusion to others. So be wary whenever you see this phrase or something similar.

You will also see the phrase 'inductive definition', and you may think this is just a variant of 'definition by induction'. In fact, it is a rather more sophisticated idea concerned with finding the least solution of a specification, and being able to describe this in a certain way. The technique is useful (and non-trivial) but its given name is nonsense. Unfortunately the terminology is used by many people some of whom ought to know better.

Exercises

1 Show how the progressive form of induction can be rephrased as a b/s form.

2 (a) Verify the explicit descriptions for the functions given in Examples 1.1(a).

(b) Consider the 3-placed functions specified by

$$f(0, y, z) = g(y, z) \qquad f(x + 1, y, z) = h(y, z, f(x, y, z))$$

for $x \in \mathbb{N}$. Here g and h are known functions. Can you find an explicit description of f in terms of g, h?

3 (a) Verify the explicit description for the fibonacci function f given in Examples 1.1(b). This involves saying what then magic number τ must be.

(b) Describe how the functions F and f are related.

4 For the function f of Examples 1.1(c) show that

$$f(x, i) = \begin{cases} 0 & \text{if } x < i \\ \frac{x!}{i!(x-i)!} & \text{if } i \leq x \end{cases}$$

for all $x, i \in \mathbb{N}$. Set up the induction carefully.

5 (a) Look up and write down the general definition of a primitive recursion which delivers a value

$$f(x, y_1, \ldots, y_s)$$

using x as the recursion variable.

(b) Rephrase this definition to deliver a value

$$f(x, p)$$

by tupling the parameters into one.

2 Setting the scene

We are concerned with ways of characterizing functions. Put differently we are concerned with ways of isolating a particular function within a whole class of functions. We will concentrate on one particular method, the use of a recursive specification. Furthermore, we will consider only a few instances of this method.

In this section we first set down some terminology, notation, and other superficial aspects. After that we will give, in Table 1, the three most common forms of specification. Once this material is understood, the section has little content beyond a few definitions.

As explained in the previous section, we are interested in a function f which may be many-placed and may require inputs of several different types. For what we do here, one of the inputs has a special role, it is the recursion variable and it is convenient to reserve 'x' for that input. So that we don't clutter up the account we bundle the other inputs into a tuple, and think of that as a single input, which we call the parameter. Thus the function f will have

$$f(x, p)$$

as a typical value, where x is the recursion variable and p is the parameter. This condensed notation is used to bring out the essential features of the analysis.

Thus we are interested in producing a function

$$\mathbb{N}, \mathbb{P} \xrightarrow{\ f\ } \mathbb{T}$$

where

- $\mathbb{N}$ is the domain of natural numbers
- $\mathbb{P}$ is a domain of parameters
- $\mathbb{T}$ is the domain of values taken by f

using various given data functions and boolean predicates. There are a few things here that need explaining. So for the next couple of pages or so (until just before Definition 2.2) that is what we will do.

We are using the word 'domain' in an informal sense. We could just as well say '*set* of natural numbers' and '*space* of parameters'. The word 'domain' has a precise meaning which agrees with the use here, and later we will have a glimpse of this more technical usage. (The meaning is not quite what you think it is.)

The type of the function f looks a little odd, doesn't it.

Given two sets $\mathbb{S}$ and $\mathbb{T}$ we may consider a function g with source $\mathbb{S}$ and target $\mathbb{T}$. Thus g is a function which consumes an input from $\mathbb{S}$ to return a value in $\mathbb{T}$. Normally we write one of

$$g : \mathbb{S} \longrightarrow \mathbb{T} \qquad\qquad \mathbb{S} \xrightarrow{\;g\;} \mathbb{T}$$

to convey this information.

Given $\mathbb{S}$ and $\mathbb{T}$ let $[\mathbb{S} \to \mathbb{T}]$ be the space, that is set, of all functions from $\mathbb{S}$ to $\mathbb{T}$ (like g). There is no reason why this space itself should not be the source or target of a function. That is, there is nor reason why a function can not consume another function or return another function. (Such higher order functions are sometimes called functionals, but that word has other meanings which can lead to confusion.)

Take another look at the type of f.

The domain $\mathbb{P}$ of parameters can be almost anything. A parameter p could be a natural number, or a pair of natural numbers, or a tuple of numbers of some kind, or even a higher order gadget. We will see examples of this later. The function f can be viewed in two ways.

$$f : \mathbb{N} \times \mathbb{P} \longrightarrow \mathbb{T} \qquad\qquad f : \mathbb{N} \longrightarrow [\mathbb{P} \to \mathbb{T}]$$

In the left-hand view the function consumes its two arguments x and p as a pair (x, p) to return its eventual value $f(x, p)$ at once. This is the view we will use in the first few examples.

In the right-hand view the function consumes its two arguments one after the other. Thus, after consuming x it returns a function

$$fx : \mathbb{P} \longrightarrow \mathbb{T}$$

which can then consume the second argument p to return fxp, the eventual value.

This process of passing between the two views is known as currying and uncurrying (after H. Curry who didn't invent it).

$$\bullet \xmapsto{\text{currying}} \bullet \qquad\qquad \bullet \xleftarrow{\text{uncurrying}} \!\!| \; \bullet$$

Usually in mathematics we don't distinguish between the two views. However, strictly speaking the two functions are different gadgets (but intimately related). Here the curried view is helpful because in a recursion it is often

useful to hide as many of the parameters as possible. This makes the various arguments and manipulations less cluttered. We will use this trick later on.

As just mentioned, strictly speaking we should use different symbols for the function to indicate which view we are taking. A distinguishing notation is

$$f_\flat : \mathbb{N} \times \mathbb{P} \longrightarrow \mathbb{T} \qquad\qquad f^\sharp : \mathbb{N} \longrightarrow (\mathbb{P} \longrightarrow \mathbb{T})$$

and then

$$f_\flat(x, p) = f^\sharp x p$$

shows how to convert from one to the other. This idea is analysed in more generality in the study of cartesian closed categories.

In these notes we will use a notation like

$$\mathbb{N}, \mathbb{P} \xrightarrow{\ f\ } \mathbb{T}$$

to indicate that the function f requires two inputs, one from $\mathbb{N}$ and one from $\mathbb{P}$, to return a value in $\mathbb{T}$. Furthermore, the function can be viewed in two ways (curried or uncurried) and we reserve the right switch between the two views whenever it is convenient to do so. We could develop some elaborate notation for this, but it is hardly worth it here.

We are interested in finding a specification, or more precisely, a recursive specification for a function f, as above. There isn't an all embracing definition of this notion, but the idea can be explained easily enough.

A specification of a function is a kind of discrete analogue of a differential equation. Thus we set down certain conditions that the function must satisfy, and then try to 'solve' this problem, that is, find a function with the required properties. In the best cases there is a unique solution. However, often it is not possible to produce an explicit description of the solution. All our knowledge of the solution must be extracted from the specification/differential equation.

We will restrict our attention to recursive specifications, that is specifications built up in certain ways, which we will expand on shortly. Given such a specification there are several questions we should investigate.

- Are there any solutions?
- Is there a unique solution and if not, is there a canonical solution?
- Are there any total solutions?
- How are the various solutions related?

The subject of Domain theory is designed, in part, to answer such questions. An introduction to this topic is given in Domains for recursion. At this point it is worth stating the crucial existence and canonicity result.

We are interested in functions

$$\mathbb{S} \xrightarrow{\ f\ } \mathbb{T}$$

where $\mathbb{S}$ and $\mathbb{T}$ are the source and the target domains. (Here 'domain' has a technical meaning.) We wish to locate such a function, or class of functions, each of which satisfies certain conditions. These functions may be partial, that is of the form

$$f : \mathbb{F} \longrightarrow \mathbb{T}$$

where $\mathbb{F}$ is a part of $\mathbb{S}$.

In the statement of the following Proposition we refer to a 'suitable' specification. This is to avoid getting into the technicalities (which wouldn't help at this stage). However, almost any specification you will ever come across, even the weirdest, is 'suitable'.

2.1 PROPOSITION. *For each 'suitable' specification of functions f, as above, there is a unique $\mathbb{D} \subseteq \mathbb{S}$ and a unique function*

$$f_D : \mathbb{D} \longrightarrow \mathbb{T}$$

where f_D satisfies the specification and every other solution extends f_D.

This function f_D is the canonical solution of the specification, and is often referred to as *the* solution. Here, in the main, we will restrict our attention to specifications that have a unique solution, and this solution is total. In the notation of the Proposition we have $\mathbb{D} = \mathbb{S}$. Furthermore, by inspecting the specification we may extract an algorithm for evaluating the function.

You will often see the phrase 'consider the function f defined by ...' where the dots are some specification. Strictly speaking this is wrong until it has been proved that the specification has a unique solution or there is a convention that we always take a canonical solution. Almost every specification we deal with here does have a unique solution, but we will try to be careful and not use the phrase 'defined by' in this way.

That is the idea of a specification, but what is a *recursive* specification?

Suppose we have a specification of a function f, as above, with typical value $f(x, p)$ where $x \in \mathbb{N}, p \in \mathbb{P}$.

In many cases the specification will have embedded within it an algorithm for evaluating f. There will be a description of how to calculate the value $f(x, p)$ in terms of the value $f(x^-, p^+)$ for some other input x^- and some variant p^+ of p. But how do we calculate $f(x^-, p^+)$? We go back to the specification to find that we first need to calculate $f(x^{--}, p^{++})$ for some input x^{--} and parameter p^{++}. We repeat this idea, passing through the specification again and again in the hope that somehow the problem is simplifying. In other words, we set

$$x(0) = x \quad x(i+1) = x(i)^- \qquad p(0) = p \quad p(i+1) = p(i)^+$$

for sufficiently many $i \in \mathbb{N}$, so that we may calculate

$$\begin{array}{ll} f(x(0), p(0)) & \text{which uses} \\ f(x(1), p(1)) & \text{which uses} \\ f(x(2), p(2)) & \text{which uses} \\ \vdots & \\ f(x(i), p(i)) & \text{which uses} \\ \vdots & \end{array}$$

and so on. In other words we first set up a regression

$$\big(x(0), p(0)\big) \mapsto \big(x(1), p(1)\big) \mapsto \big(x(2), p(2)\big) \mapsto \cdots$$

by repeatedly passing through the recursion step. Once this regression bottoms out (that is, terminates) we can work back up the chain to produce the required value.

Of course, we need some guarantee that this regression does stop. As an attempt to do that the specification will give a set $\mathbb{B}$ of inputs x together with information

$$f(x, p) = \text{ explicit definition}$$

for those x. The regression will bottom out when $x(i) \in \mathbb{B}$.

This is the idea of a recursion over x. In the simplest cases we have $x^- < x$ and the recursion bottoms out at 0. Initially we will concentrate on these simple cases.

We are almost in a position to give the first definition (of a class of recursive specifications). We still need a useful bit of terminology and notation.

A specification produces a function f out of other functions $g, h, k \ldots$ which we assume are given and we know all about. We call these 'input' or 'known' functions the data functions to distinguish them from the 'output' or 'unknown' function f.

Finally, a piece of notation that is useful here (but not standard). This concerns '$(\cdot)'$' which we use use in two ways. For each $x \in \mathbb{N}$ we write x' for the successor of x, that is $1 + x$. Notice that x' is the next most complicated number after x. For a set $\mathbb{S}$ we write $\mathbb{S}'$ for the type $(\mathbb{S} \longrightarrow \mathbb{S})$ of functions from $\mathbb{S}$ to itself. Notice that $\mathbb{S}'$ can be thought of as the next most complicated set after $\mathbb{S}$. Both of these notations can be iterated. Thus we have

$$x' = 1 + x \quad x'' = 1 + x' = 2 + x \quad x''' = 1 + x'' = 3 + x \quad \cdots$$

and

$$\mathbb{S}' = (\mathbb{S} \longrightarrow \mathbb{S}) \quad \mathbb{S}'' = (\mathbb{S}' \longrightarrow \mathbb{S}') = ((\mathbb{S} \longrightarrow \mathbb{S}) \longrightarrow (\mathbb{S} \longrightarrow \mathbb{S})) \quad \cdots$$

and so on.

At last we can set up the first few forms of recursions.

Head

$$\begin{array}{ll} f(0\,,p) = gp & f(x,p) = \textbf{\textit{if}}\ x=0\ \ \textbf{\textit{then}}\ \ gp \\ f(x',p) = h(x,p,t) & \qquad\qquad\qquad\qquad \textbf{\textit{else}}\ \ h(x,p,t) \\ \text{where } t = f(x,p) & \qquad\qquad\qquad\qquad \textbf{\textit{where}}\ \ t \ \ = f(x^-,p) \\ & \qquad\qquad\qquad\qquad\qquad\qquad x^- = x-1 \\ & \qquad\qquad \textbf{\textit{fi}} \end{array}$$

Body

$$\begin{array}{ll} f(0\,,p) = gp & f(x,p) = \textbf{\textit{if}}\ x=0\ \ \textbf{\textit{then}}\ \ gp \\ f(x',p) = h(x,p,t) & \qquad\qquad\qquad\qquad \textbf{\textit{else}}\ \ h(x,p,t) \\ \text{where } t = f(x,p^+) & \qquad\qquad\qquad\qquad \textbf{\textit{where}}\ \ t \ \ = f(x^-,p^+) \\ \qquad\ p^+ = k(x,p) & \qquad\qquad\qquad\qquad\qquad\qquad x^- = x-1 \\ & \qquad\qquad\qquad\qquad\qquad\qquad p^+ = k(x^-,p) \\ & \qquad\qquad \textbf{\textit{fi}} \end{array}$$

Tail

$$\begin{array}{ll} f(0\,,p) = gp & f(x,p) = \textbf{\textit{if}}\ x=0\ \ \textbf{\textit{then}}\ \ gp \\ f(x',p) = f(x,p^+) & \qquad\qquad\qquad\qquad \textbf{\textit{else}}\ \ f(x^-,p^+) \\ \text{where } p^+ = k(x,p) & \qquad\qquad\qquad\qquad \textbf{\textit{where}}\ \ p^+ = k(x,p) \\ & \qquad\qquad\qquad\qquad\qquad\qquad x^- = x-1 \\ & \qquad\qquad \textbf{\textit{fi}} \end{array}$$

Table 1: The head, body, and tail recursions in base/step and conditional form

---

2.2 DEFINITION. Assume we are given three data functions

$$\mathbb{P} \xrightarrow{\ g\ } \mathbb{T} \qquad \mathbb{N},\mathbb{P},\mathbb{T} \xrightarrow{\ h\ } \mathbb{T} \qquad \mathbb{N},\mathbb{P} \xrightarrow{\ k\ } \mathbb{P}$$

for some $\mathbb{P},\mathbb{T}$. We combine these in various ways to specify a target function

$$\mathbb{N},\mathbb{P} \xrightarrow{\ f\ } \mathbb{T}$$

by recursion over the left-most argument x.

We use three different forms of recursion. These may be called

Head Body Tail

recursion to produce f. Each has a

base/step conditional

version as given in Table 1. In each case the base/step version is on the left and the conditional version is on the right. ■

The terminology 'tail' recursion is standard. The other terminology is not. (A formal definition of tail recursion is rarely given. An explanation of why the word is used is never given.)

The b/s version of head recursion is sometime referred to as primitive recursion. However, this is correct *only* when the parameter p is a list of natural numbers. In this restricted case the body form is known as primitive recursion with variation of parameters. Later we will see how a head recursion (with more exotic parameters) can far out jump the primitive recursive functions.

Notice that if one or other of the data functions k, h is a projection with

$$k(x,p) = p \qquad h(x,p,t) = t$$

then the Body recursion becomes a

Head Tail

recursion, respectively. It is these connections that suggest the terminology.

The differences between the b/s version and the conditional version are mostly cosmetic, but sometimes one version is more convenient than the other. Each can be rephrased in the other form, but this requires a slight modification of the data functions.

A word about the notation used in the conditional form is in order. Consider the body recursion. This is more commonly written

$$f(x,p) = \begin{cases} gp & \text{if } x = 0 \\ h(x,p,t) & \text{if } x \neq 0 \quad \text{where } \cdots \end{cases}$$

but some people insist on the linear form. It is as well to get used to it for you never know when it might appear. The fancy fount ***of this kind*** makes the construction look like a pseudo-program. The '***fi***' at the end is a punctuation symbol to indicate that the clause opened by the initial '***if***' has now closed. You will see why this is needed in Examples 2.6 and 2.7.

There are various particular cases of these forms. For instance, the values of the data function h may depend only on some of the inputs. Hilbert observed that, by currying, many specification could be knocked into the form

$$f0 = \text{explicit value} \qquad fx' = h(x, fx)$$

for a suitable data function h. This was later used to great affect by Gödel. (The catch is that higher order gadgets are needed to hide the parameters.)

To understand some of the differences between these forms of recursion you should work out how the specification can be used to evaluate the function. Go through the general regression method suggested earlier. You will see that some forms are easier to use than others.

There are many other forms of recursion, and it is useful to see some of these, even though we will not analyse all of them.

In Definition 2.2 the recursion variable x changes by at most 1 for each pass through the recursion. Sometimes this is not convenient.

2.3 DEFINITION. Assume we are given three data functions

$$\mathbb{P} \xrightarrow{\quad g \quad} \mathbb{T} \qquad \mathbb{N}, \mathbb{P}, \mathbb{T} \xrightarrow{\quad h \quad} \mathbb{T} \qquad \mathbb{N}, \mathbb{P} \xrightarrow{\quad k \quad} \mathbb{P}$$

and wish to specify a target function

$$\mathbb{N}, \mathbb{P} \xrightarrow{\quad f \quad} \mathbb{T}$$

by recursion over the left-most argument x. We assume also a descent function $d \in \mathbb{N}'$ which will control the recursion. This must satisfy

$$dx < x$$

for each $0 \neq x \in \mathbb{N}$, and it is often convenient to set $d0 = 0$.

With these

$$f(x,p) = \textit{\textbf{if}}\ x = 0\ \textit{\textbf{then}}\ gp\ \textit{\textbf{else}}\ h(x,p,t)\ \textit{\textbf{where}}\ \left\{\begin{array}{l} t\ = f(x^-, p^+) \\ x^- = dx \\ p^+ = k(x^-, p) \end{array}\right\}\ \textit{\textbf{fi}}$$

is an instance of descent body recursion. ■

Body recursion is just the case where d is the predecessor function.

Earlier we spoke about the difference between *specifying* and *defining* a function. For all of the recursions we have seen so far there is a unique solution, and this is total. For these the difference is little more than pedantry. However, with more general recursions the difference becomes significant.

2.4 EXAMPLE. Suppose we have three data functions

$$\mathbb{S} \xrightarrow{\quad g \quad} \mathbb{T} \qquad \mathbb{S}, \mathbb{T} \xrightarrow{\quad h \quad} \mathbb{T} \qquad \mathbb{S} \xrightarrow{\quad k \quad} \mathbb{S}$$

together with a subset $\mathbb{B} \subseteq \mathbb{S}$. We use these to specify a function

$$\mathbb{S} \xrightarrow{\quad f \quad} \mathbb{T}$$

by

$$fs = \textit{\textbf{if}}\ s \in \mathbb{B}\ \textit{\textbf{then}}\ gs\ \textit{\textbf{else}}\ (hs)(fs^-)\ \textit{\textbf{where}}\ s^- = ks\ \textit{\textbf{fi}}$$

where s ranges over $\mathbb{S}$. ■

All the previous examples can be knocked into this shape (and this form is far more general than these cases).

Here we have used h in curried form (and so have hidden any parameters that may be around). Notice also that we have imposed no conditions on k. It is this aspect that can cause problems. Let's look at an example which has several demons, some hidden and some not.

2.5 EXAMPLE. Consider a function

$$\mathbb{R} \xrightarrow{\ f\ } \mathbb{R}$$

specified by

$$fx = \textit{\textbf{if}}\ x = 0\ \textit{\textbf{then}}\ 0\ \textit{\textbf{else}}\ 1 + f^2(x-1)\ \textit{\textbf{fi}}$$

where x ranges over $\mathbb{R}$. What can such a function look like?

Try to calculate a few values of f. You will find that $fx = x$ must hold for all $x \in \mathbb{N}$, but there seems no way of getting at $x \in \mathbb{R} - \mathbb{N}$. In fact, this function $\mathbb{N} \longrightarrow \mathbb{R}$ thought of as a *partial* function on $\mathbb{R}$, is a solution of the specification.

There are many other solutions.

Consider any subset $A \subseteq [0,1)$ with $0 \in A$. Consider any function $l \in A'$ with $l0 = 0$ and $l^2 = l$. For each $x \in \mathbb{R}$ we have

$$x = \lfloor x \rfloor + \langle x \rangle \qquad \text{where } 0 \leq \langle x \rangle < 1$$

and $\lfloor x \rfloor$ is the integer floor of x. It can be checked that

$$fx = \begin{cases} \lfloor x \rfloor + l\langle x \rangle \text{ if } \langle x \rangle \in A \\ \text{undefined if } \langle x \rangle \notin A \end{cases}$$

meets the specification.

In particular, taking $A = [0,1)$ we see there are infinitely many total functions which meet the specification. Taking smaller sets A we see there infinitely many partial functions which meet the specification. ■

For the simpler recursive specifications, including the most of the ones we will look at, it can be shown that there is exactly one function which meets it, and this function is total. Usually this is done using the associated form of induction, and often without knowing very much about the unique solution. This is the case with the recursions of Definitions 2.2 and 2.3. In such a case we can think of the specification as defining the function (once we have proved the required uniqueness and existence).

To conclude this section let's look at a couple of specifications which illustrate that even simple recursions can require some complicated analysis. That is, for these two examples, it is the form of the recursion that is simple, but the analysis of what it produces is not.

2.6 EXAMPLE. Consider the 2-placed function

$$\mathbb{N}, \mathbb{N} \xrightarrow{\ f\ } \mathbb{N}$$

specified by

$$\begin{array}{llll} f(x,i) = \textit{\textbf{if}}\ i=0 & \textit{\textbf{then}} & x & \\ & \textit{\textbf{else}} & \textit{\textbf{if}}\ x \le 100 & \textit{\textbf{then}}\ \ f(x+11, i+1) \\ & & & \textit{\textbf{else}}\ \ f(x-10, i-1) \\ & & \textit{\textbf{fi}} & \\ \quad \textit{\textbf{fi}} & & & \end{array}$$

(for $i, x \in \mathbb{N}$). ■

In this example there are two recursion variables with two conditionals, one nested inside the other. Both recursion variables can increase or decrease on a pass through the recursion. The example speaks of '*the* 2-placed function' but it is far from clear that the specification has a unique solution (or any solution), and it is even less clear that this function is total. In fact, this specifies a rather simple function in a daft way.

2.7 EXAMPLE. Consider the 2-placed function

$$\mathbb{N}, \mathbb{N} \xrightarrow{\ f\ } \mathbb{N}$$

specified by

$$\begin{array}{llll} f(x,y) = \textit{\textbf{if}}\ x \le 1 & \textit{\textbf{then}} & y & \\ & \textit{\textbf{else}} & \textit{\textbf{if}}\ x \text{ is even} & \textit{\textbf{then}}\ \ f(x/2, y+1) \\ & & & \textit{\textbf{else}}\ \ f(3x+1, y+1) \\ & & \textit{\textbf{fi}} & \\ \quad \textit{\textbf{fi}} & & & \end{array}$$

(for $x, y \in \mathbb{N}$). ■

Here there is just one recursion variable x, but this can increase or decrease on each pass through the recursion. The parameter y does little more than count the number of passes through the recursion. This specification does have a unique solution, but nobody knows if this is total. That is, nobody knows if the algorithm embedded in the specification does terminate for every input x.

Exercises

6 Consider the three functions

$$\mathbb{N}, \mathbb{N}, \mathbb{N} \xrightarrow{\ H, B, T\ } \mathbb{N}$$

specified recursively by

$$\begin{array}{ll} H(0\ , y, z) = z & H(x', y, z) = H(x, y, z) \times (y + x) \\ B(0\ , y, z) = z & B(x', y, z) = y \times B(x, y', z) \\ T(0\ , y, z) = z & T(x', y, z) = T(x, y', zy) \end{array}$$

for $x, y, z \in \mathbb{N}$.

(a) Fit each of these specifications into the forms of Definition 2.2. In particular, show how the parameters $p = (y, z)$ are handled.

(b) Use these specifications to organize informal algorithms to calculate

$$H(17, y, z) \qquad B(17, y, z) \qquad T(17, y, z)$$

(for arbitrary $y, z \in \mathbb{N}$). Which of the three forms of recursion leads to more efficient computations?

(c) Show that

$$H(x', y, z) = y \times H(x, y', z)$$

holds for all $x, y, z \in \mathbb{N}$, and hence show that $H = B$. This requires two inductions. You should state carefully both induction hypotheses.

(d) Show that

$$B(x', y, z) = B(x, y, z) \times (y + x)$$

holds for all $x, y, z \in \mathbb{N}$, and hence show that $H = B$.

(e) How do the two proofs of $H = B$ given by (c,d) differ?

(f) Show that

$$T(x, y + u, H(u, y, z)) = H(x + u, y, z)$$

holds for all $u, x, y, z \in \mathbb{N}$, and hence show that $H = T$.

State carefully any induction hypothesis that you use.

(g) Give an explicit description of the common function $H = B = T$.

7 Given an arbitrary function ϕ, as to the left

$$\mathbb{S}, \mathbb{T} \xrightarrow{\ \phi\ } \mathbb{T} \qquad\qquad \mathbb{N}, \mathbb{S}, \mathbb{T} \xrightarrow{\ H,T\ } \mathbb{T}$$

we may specify two functions H, T, as to the right, by

$$\begin{array}{ll} H(0, s, t) = t & T(0, s, t) = t \\ H(x', s, t) = \phi(s, t^+) & T(x', s, t) = T(x, s, t^+) \\ \text{where } t^+ = H(x, s, t) & \text{where } t^+ = \phi(s, t) \end{array}$$

(for $x \in \mathbb{N}, s \in \mathbb{S}, t \in \mathbb{T}$).

Organize a proof that $H = T$ and find an explicit description of this common function.

8 For each of the head, body, tail recursion schemes, show how to rephase one form (b/s or conditinal) as the other form. This involves slightly modifying the data functions.

> Each of the simpler forms recursive specification over $\mathbb{N}$ has precisely one solution. The uniqueness part becomes more interesting if it is rephrased in terms of equivalences. The next exercise shows a way of doing this.

9 Consider the two functions

$$\mathbb{N},\mathbb{P} \xrightarrow{f_1} \mathbb{T} \qquad\qquad \mathbb{N},\mathbb{P} \xrightarrow{f_2} \mathbb{T}$$

specified by body recursion (in b/s form) from the data functions

$$\begin{array}{rcl} \mathbb{P} \xrightarrow{g_1} \mathbb{T} & \qquad & \mathbb{P} \xrightarrow{g_2} \mathbb{T} \\ \mathbb{N},\mathbb{P} \xrightarrow{k_1} \mathbb{P} & & \mathbb{N},\mathbb{P} \xrightarrow{k_2} \mathbb{P} \\ \mathbb{N},\mathbb{P},\mathbb{T} \xrightarrow{h_1} \mathbb{T} & & \mathbb{N},\mathbb{P},\mathbb{T} \xrightarrow{h_2} \mathbb{T} \end{array}$$

respectively. Suppose $\mathbb{P}$ and $\mathbb{T}$ carry equivalence relations $\sim$ and $\approx$, respectively, where

$$\begin{array}{ll} (\gamma) & p_1 \sim p_2 \Longrightarrow gp_1 \approx gp_2 \\ (\kappa) & p_1 \sim p_2 \Longrightarrow k_1(x,p_1) \sim k_2(x,p_2) \\ (\eta) & (p_1 \sim p_2) \wedge (t_1 \approx t_2) \Longrightarrow h_1(x,p_1,t_1) \approx h_2(x,p_2,t_2) \end{array}$$

hold for all $p_1, p_2 \in \mathbb{P}, x \in \mathbb{N}, t_1, t_2 \in \mathbb{T}$. Using induction over $\mathbb{N}$ show that

$$(\forall x : \mathbb{N})(\forall p_1, p_2 : \mathbb{P})[(p_1 \sim p_2) \Longrightarrow f_1(x,p_1) \approx f_2(x,p_2)]$$

holds. Set up the induction hypothesis carefully.

10 Using the mod2 and div2 functions consider the function

$$\mathbb{N},\mathbb{N},\mathbb{N} \xrightarrow{\text{boris}} \mathbb{N}$$

specified by

$$\text{boris}(x,y,z) = \begin{cases} \textit{if} \quad x = 0 \;\textit{then}\; z \;\textit{else}\; \text{boris}(x^-, y^+, z^+) \\ \qquad\qquad\qquad \textit{where} \left\{ \begin{array}{ll} i & = z \text{ mod2} \\ z^+ & = iy + z \\ y^+ & = 2y \\ x^- & = x \text{ div2} \end{array} \right\} \\ \textit{fi} \end{cases}$$

(for $x, y, z \in \mathbb{N}$). Calculate a few values of boris and then verify an explicit description

$$\text{boris}(x,y,z) = \cdots$$

of boris. State carefully any induction hypotheses that you use.

11 Consider the specification

$$\text{blaise}(0,y) = 1 = \text{blaise}(x,0) \qquad \text{blaise}(x',y') = \text{blaise}(x',y) + \text{blaise}(x,y')$$

of a function

$$\mathbb{N}, \mathbb{N} \xrightarrow{\text{blaise}} \mathbb{N}$$

(where $x, y \in \mathbb{N}$).

Show there is at most one such function blaise. Set up the induction hypothesis very carefully. Finally describe the unique solution of this specification.

12 Consider the specification of Example 2.6. Consider also the function

$$\mathbb{N}, \mathbb{N} \xrightarrow{d} \mathbb{N}$$

given by

$$d(x, i) = \begin{cases} 21i & \text{if } 111 < x \\ 21i + 2(111 - x) & \text{if } x \leq 111 \end{cases}$$

for $x, i \in \mathbb{N}$.

(a) Show that for each $x, i \in \mathbb{N}$ we have

$$d(x, i) - 21 \leq d(x^+, i^+) < d(x, i)$$

where x^+, i^+ are the next inputs needed after one pass through the recursion. Hence show that the specification has a unique solution, and this function is total.

(b) Show that unique solution is given by

$$f(x, 0) = x \qquad f(x, i) = \begin{cases} x - 10i & \text{if } 90 + 10i \leq x \\ 91 & \text{if } x < 90 + 10i \end{cases}$$

for $i \neq 0$.

13 Consider the specification of Example 2.7.

(a) Show that the specification has precisely one solution f and that

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

for all $x, y \in \mathbb{N}$ where the function is defined.

(b) Calulate the values $f(x, 0)$ for the range $1 \leq x \leq 50$ (say).

(c) Draw up a table of values

$$x \quad s \quad \underline{x} \quad l \quad \overline{x}$$

where x is the input in the chosen range, s is the least number of passes through the recursion before the next input $\underline{x}$ is obtained in the chosen range, l is the total number of passes through the recursion before the value of $f(x, 0)$ is obtained, and $\overline{x}$ is the largest input needed in this terminating evaluation.

(d) Show that the function is total.

3 The development and complexity of arithmetic

The title of this section could mean several things depending on who you are. So we will begin with an explanation.

The word 'arithmetic' has several meanings, even for what we do here. The most common meaning is concerned with what Lewis Carroll's Mock Turtle called Ambition, Distraction, Uglification, and Derision; and what the populace today thinks is 'mathematics'. More precisely, it is concerned with the simple functions on $\mathbb{N}$, say those generated from $+$ and $\times$ using fairly simple methods of construction. If we want to go beyond that then we could add $\bullet^{\bullet}$, exponentiation. At another extreme we could think of peano arithmetic, the first order theory which Gödel showed is incomplete and undecidable. We are interested in both these extremes and the stuff in between (and stuff beyond that).

(The curious notation used above for exponentiation is a minor illustration of the 'complexity' of functions. There are common symbols for addition and multiplication, that is $+$ and $\times$, but none for exponentiation. This is because exponentiation is 'more complicated' than the other two operations. We won't persists with this notation.)

By 'development' we do not mean the historical development. Rather we mean something like the formal development of the material from a few basic principles. The kind of thing that has to be done as a preamble to Gödel's proof. The best, or perhaps most detailed, account of this that I know of is in [6] by Mendelson. We will refer to that account later, so perhaps to you should look at it. The copy of [6] I have was published in 1964. Since then it has been revised, so there may be some discrepancies in my references to particular pages or results. (I bet you didn't know that a certain T.G. McLaughlin once wrote a paper on a topic related to this material called 'A muted variation on a theme of Mendelson'.)

We can think of the arithmetic of Carroll as embedded in that of Peano. It it is clear that one is much simpler than the other. To develop the simpler version we need far less power. In technical terms we need only simple instances of the induction principles available in peano arithmetic. This is what 'complexity' is getting at. Intuitively we can see that the operations

Successor   Addition   Multiplication   Exponentiation $\cdots$

are getting progressively more complicated, harder to handle, and we suspect that things could get a lot more complicated. An analysis of 'complexity' tries to make this intuitive idea more precise.

Let's look at how a formal development of arithmetic might begin, as set out by Mendelson in chapter 3 of [6]. We will also make use of page 194 of the book [3] by Enderton.

We first need to set up the appropriate formal language. (This is the point that Peano stressed. The axioms themselves are essentially due to Dedekind.)

We certainly need a symbol to name zero. We will use $\mathsf{0}$ (in sans serif) so as not to confuse it with $0 \in \mathbb{N}$. Mendelson doesn't emphasize this distinction, but Enderton does. We also need a name for the successor operation. Mendelson writes this as $(\cdot)'$ which is not a good idea. (Besides, we have an informal usage of this.) We follow Enderton and write S for this operation.

With these two symbols we can produce a name for each natural number.

3.1 DEFINITION. The numerals are generated from primitive symbols $\mathsf{0}$ and S.

- $\mathsf{0}$ is a numeral.
- If x is a numeral then so is $\mathsf{S}x$.

Let $\ulcorner\mathbb{N}\urcorner$ be the set of numerals. ■

Each numeral is a piece of syntax. It is a list

$$\mathsf{S}\cdots\mathsf{S0}$$

of Ss ending with $\mathsf{0}$. Of course, this list of Ss may be empty. This is the name of the corresponding number of Ss. We write

$$\ulcorner m\urcorner \text{ for the name of } m$$

so that

$$\ulcorner 0\urcorner \text{ is } \mathsf{0} \qquad \ulcorner 1+m\urcorner \text{ is } \mathsf{S}\ulcorner m\urcorner$$

for each $m \in \mathbb{N}$. In particular, we have

$$\ulcorner\mathbb{N}\urcorner = \{\ulcorner m\urcorner \mid m \in \mathbb{N}\}$$

and we see the pedantic distinction. Given a numeral y we write

$$\mathsf{S}^m y \text{ for } \mathsf{S}\cdots\mathsf{S}y$$

where there are m number of Ss before y. Thus

$$\ulcorner m\urcorner = \mathsf{S}^m\mathsf{0}$$

and, more generally, we have

$$\mathsf{S}^m\ulcorner n\urcorner = \ulcorner n+m\urcorner$$

for $m, n \in \mathbb{N}$. Here the equality means that the two pieces of syntax are exactly the same.

We could extend this signature to a full first order language and write down the appropriate axioms for the theory of $(\mathbb{N}, 0, S)$ (where here S is the actual successor operation). It turns out that this theory is not very complicated. It

is finitely axiomatizable, complete, and decidable. In fact, for this theory there is a fairly straight forward method of eliminating quantifiers. (Any reasonably balanced course on mathematical logic will contain this result somewhere.)

This theory is 'trivial' because the language is so feeble it can hardly say anything. (Remember Peano's point.) We need a more powerful language, and clearly we need to incorporate addition, multiplication, and perhaps exponentiation. Let us write '$+$' and '$\times$' for the formal addition and multiplication. Furthermore, we will write these as infix notation, in the usual way. (This is not a good idea on two counts, but it will do for the time being.) We also introduce a notation for exponentiation, which Mendelson doesn't do but Enderton does.

These three operations need controlling axioms. The standard ones are

$(S5)$	$A1.$	$y + \mathsf{0} = y$	$(S6)$	$A2.$	$y + \mathsf{S}x = \mathsf{S}(y + x)$
$(S7)$	$M1.$	$y \times \mathsf{0} = \mathsf{0}$	$(S8)$	$M2.$	$y \times \mathsf{S}x = y \times x + y$
	$E1.$	$y^{\mathsf{0}} = \ulcorner 1 \urcorner$		$E2.$	$y^{\mathsf{S}x} = y^x \times y$

where the labels are the ones use by Mendelson and Enderton, respectively. In a full and formal development of peano arithmetic many consequences of these axioms are derived (such that addition and multiplication is associative and commutative, the distributive law, the exponent laws, and so on). A lot of this is done by Mendelson but omitted by Enderton.

We don't want to follow that path here, but we do want to follow a similar, and almost parallel, path. We are not concerned here with deriving properties of operations on natural numbers (such as whether or not a certain operation is commutative), but we do want to look at the efficiency of certain algorithms.

Each of the three pairs of axioms is a primitive recursive specification of the operation (in terms of earlier operations). In particular, each of the three pairs has an embedded algorithm for evaluating that operation. Are these algorithms any good? To answer this question, or rather make it more precise, we will reformulate each algorithm in a 'better' notation and adjoin a competing algorithm. We will then compare the three pairs of algorithms. In each case we will keep in mind several questions. How efficient is that algorithm? What is the cost of running that algorithm? Is there a better evaluation algorithm?

To answer questions like these we have to take a little care and bring out a distinction not normally visible in mathematics. However, once the point is understood we can conform to the standard practice and be more relaxed about the distinction.

The natural numbers, the members of $\mathbb{N}$, are abstract entities without any physical existence. However, to run an algorithm, that is to perform a calculation, we need to manipulate physical entities which represent the numbers involved (and perhaps other abstract entities). In short, an algorithm operates with *names* for natural numbers, not the numbers themselves.

In everyday life we use the decimal notation to name numbers. Centuries ago we would have used, perhaps, the roman numerals, and earlier even cruder

notation systems. The form of the names employed has an impact on the efficiency of the algorithm. To illustrate this evaluate

$$\text{CMIL} \times \text{DCCXLVI} \qquad 949 \times 746$$

in the indicated notation. After that you might try to write down the full algorithm for multiplication for the two systems.

(You often see the copyright date of a television programme written in roman numerals. This is done precisely to make it more difficult to read, and so realized that the stuff is years old. At least that was the case until MM came along. The accounting department of the Treasury in Britain is called the Exchequer because until a couple of centuries ago the calculations were done by moving around piles of counters on a chequered board. In many parts of the world the abacus is still used for day to day transactions.)

Here we will use the most primitive was of naming numbers, by the corresponding number of marks on paper. In short, we will keep the score. We will calculate directly with numerals, which is why we introduced the numerals.

3.2 DEFINITION. Consider the following algorithms which operate on pairs on numerals.

$$\begin{array}{rlrl} A_l(\mathsf{0},y) &= y & A_r(\mathsf{0},y) &= y \\ A_l(\mathsf{S}x,y) &= \mathsf{S}A_l(x,y) & A_r(\mathsf{S}x,y) &= A_r(x,\mathsf{S}y) \\[1ex] M_l(\mathsf{0},y) &= \mathsf{0} & M_r(\mathsf{0},y) &= \mathsf{0} \\ M_l(\mathsf{S}x,y) &= A(M_l(x,y),y) & M_r(\mathsf{S}x,y) &= A(y,M_r(x,y)) \\[1ex] E_l(\mathsf{0},y) &= \ulcorner 1 \urcorner & E_r(\mathsf{0},y) &= \ulcorner 1 \urcorner \\ E_l(\mathsf{S}x,y) &= M(E_l(x,y),y) & E_r(\mathsf{S}x,y) &= M(y,E_r(x,y)) \end{array}$$

Here x, y range over $\ulcorner \mathbb{N} \urcorner$, so each must be a numeral. In the middle algorithms A is either A_l or A_r, and in the bottom algorithm M is either M_l or M_r (so in all there are 14 different algorithms). ■

How do we use these algorithms? Suppose we want to evaluate $3 + 4$ by running either A_l or A_r. We use the numerals $\ulcorner 3 \urcorner$ and $\ulcorner 4 \urcorner$, and must reduce

$$A_l(\ulcorner 3 \urcorner, \ulcorner 4 \urcorner) \qquad A_r(\ulcorner 3 \urcorner, \ulcorner 4 \urcorner)$$

to a numeral (which, because of out superior intelligence, we know should be $\ulcorner 7 \urcorner$). Since $\ulcorner 3 \urcorner = \mathsf{S}\ulcorner 2 \urcorner$ this starting compound can be transformed to

$$\mathsf{S}A_l(\ulcorner 2 \urcorner, \ulcorner 4 \urcorner) \qquad A_r(\ulcorner 2 \urcorner, \mathsf{S}\ulcorner 4 \urcorner)$$

respectively. We now attack the

$$A_l(\ulcorner 2 \urcorner, \ulcorner 4 \urcorner) \qquad A_r(\ulcorner 2 \urcorner, \ulcorner 5 \urcorner)$$

by a second call on the algorithm. In this way we call on the two clauses to generate a sequence of transitions

$$A_l(\ulcorner 3 \urcorner, \ulcorner 4 \urcorner) \longrightarrow \mathsf{S}A_l(\ulcorner 2 \urcorner, \ulcorner 4 \urcorner) \longrightarrow \mathsf{SS}A_l(\ulcorner 1 \urcorner, \ulcorner 4 \urcorner) \longrightarrow \mathsf{SSS}A_l(0, \ulcorner 4 \urcorner) \longrightarrow \mathsf{S}^3 \ulcorner 4 \urcorner)$$

$$A_r(\ulcorner 3 \urcorner, \ulcorner 4 \urcorner) \longrightarrow A_r(\ulcorner 2 \urcorner, \mathsf{S}\ulcorner 4 \urcorner) \longrightarrow A_r(\ulcorner 1 \urcorner, \mathsf{SS}\ulcorner 4 \urcorner) \longrightarrow A_r(\ulcorner 0 \urcorner, \mathsf{SSS}\ulcorner 4 \urcorner) \longrightarrow \mathsf{S}^3 \ulcorner 4 \urcorner)$$

to name the correct result, since $\mathsf{S}^3\ulcorner 4 \urcorner) = \ulcorner 7 \urcorner$.

We may define the cost of a run of one of these algorithms as the number of calls on the clauses, which is equivalent to the number of transitions made to achieve the end result. Using this measure, which of A_l or A_r is cheaper to run? As the example suggests, and as can be proved, they have the same cost. Can you put a figure on that cost?

Next consider the four algorithms M_l, M_r. Can guess what is going to happen? Suppose we want to evaluate 3×4 using M_r. We must reduce

$$M_l(\ulcorner 3 \urcorner, \ulcorner 4 \urcorner) \qquad M_r(\ulcorner 3 \urcorner, \ulcorner 4 \urcorner)$$

to a numeral (which we know will be $\ulcorner 12 \urcorner$). We have a transition

$$M_l(\ulcorner 3 \urcorner, \ulcorner 4 \urcorner) \longrightarrow A(M_l(\ulcorner 2 \urcorner, \ulcorner 4 \urcorner), \ulcorner 4 \urcorner) \qquad M_r(\ulcorner 3 \urcorner, \ulcorner 4 \urcorner) \longrightarrow A(\ulcorner 4 \urcorner, M_r(\ulcorner 2 \urcorner, \ulcorner 4 \urcorner)$$

so at some stage we must invoke the algorithm A; the one we choose to use or are told to use by the algorithm for M. In this case it doesn't matter too much, for they both cost the same to run. However, before we can do that we *must* reduce the inner compound $M(\ulcorner 2 \urcorner, \ulcorner 4 \urcorner)$ to a numeral, for the algorithm A insists that both inputs are numerals. In this way we generate a sequence of transitions as in Table 2. Both of these produce the correct result but get there in different ways. There is a bit in the middle where M_l takes one step but M_r takes five steps. Also the end phases are not the same length, but in the end both algorithms take the same number of steps. However, this instance is not typical.

You should try a few more examples to work out what is going on. You will find that one algorithm is better than the other. In fact, M_l is quadratically faster than M_r. Why do you think this is?

The message of this example is that the very form of a specification can effect the efficiency of evaluating the specified function.

When we turn to E_l and E_r these kind of problems become worse. Even if we choose the more efficient M to invoke there is a big difference in the cost of running the two algorithms.

Different algorithms for the same function can have very different costs. Even when both algorithms look reasonable. (Clearly, any algorithm can be made less efficient by inserting a lot of irrelevant sidetracking.) As you can imagine, there are some subtleties in designing algorithms over and above finding a specification for the corresponding function. Even something as trivial as multiplication has some hidden demons.

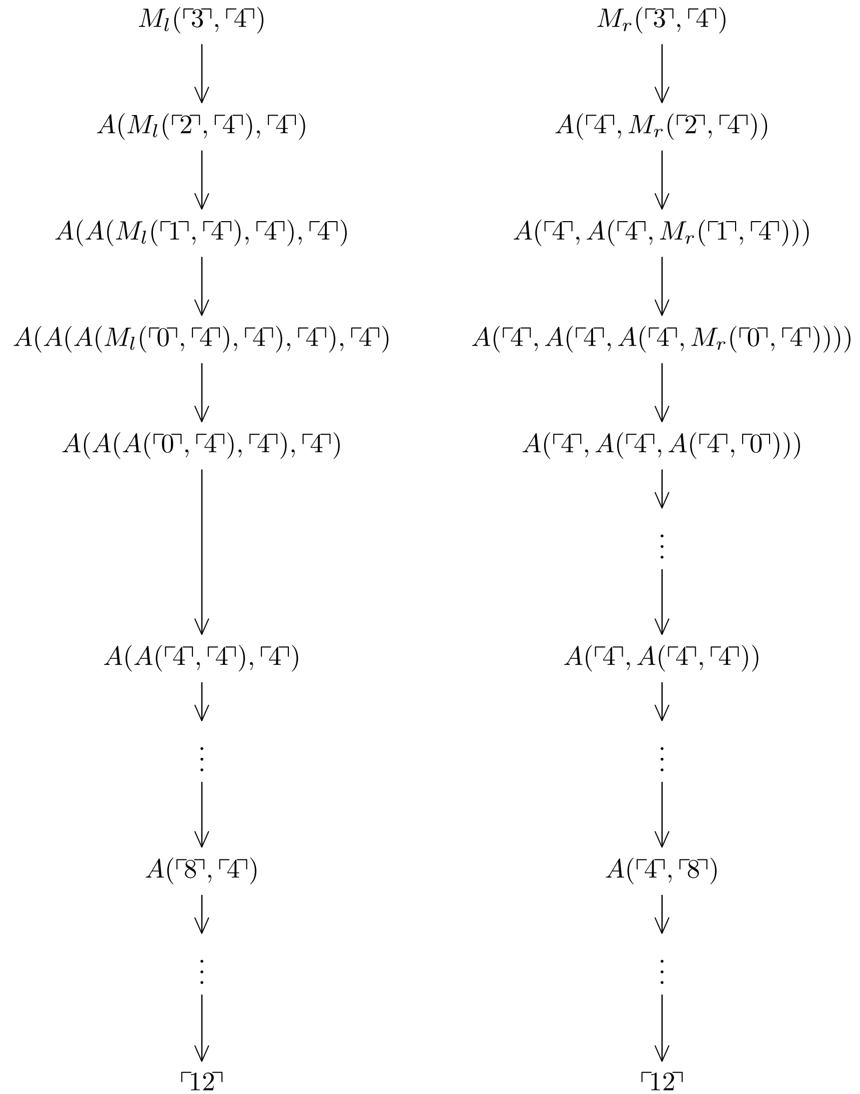

Table 2: A parallel sequence of transition

We are not going to pursue this idea. It leads to the topic of computational complexity and, on the whole, develops methods and results that are quite different from those we look at here.

The question we will try to throw some light on is: Why is one function inherently more complicated than some other? The suggested answer we will try to push is that when whole families of functions are generated in some uniform fashion some functions are always generated later than others (because they depend on the others). We will also try to use this idea to develop a measure of 'inherent complexity'. Of course, we will not go very far into this topic.

Let's begin at the beginning.

The simplest kind of recursion is iteration. Consider a head recursion, as in Table 1, and suppose the data function h doesn't depend on the inputs x and p. Thus we have a function $h \in \mathbb{T}'$ for some set $\mathbb{T}$. Such a function can be composed with itself, and by repeating this we obtain its iterates. Formally we generate the functions h^x for $x \in \mathbb{N}$ by

$$h^0 = id_{\mathbb{T}} \qquad h^{x'} = h \circ h^x$$

where $id_{\mathbb{T}}$ is the identity function on $\mathbb{T}$ and $\cdot \circ \cdot$ indicates function composition.

Many of the more common functions of arithmetic can be generated by repeated iteration.

3.3 DEFINITION. Let $S : \mathbb{N}'$ be the successor function on $\mathbb{N}$, and set

$$Ayx = S^x y \qquad Myx = (Ay)^x 0 \qquad Eyx = (My)^x 1 \qquad Bzyx = (Ey)^x z$$

to produce three 2-placed functions A, M, E and a 3-placed function B, all in curried form ■

It doesn't take to long to see that

$$Ayx = y + x \qquad Myx = y \times x \qquad Eyx = y^x$$

for all $x, y \in \mathbb{N}$. The more restricted sense of 'arithmetic' is concerned with the analysis of functions around this complexity.

Notice how the curried view hints that in the progression

$$S \longmapsto A \longmapsto M \longmapsto E \longmapsto B$$

there is a measure of complexity. In a sense these functions have measure 0,1,2,3,4, respectively. How can we make that precise?

We can convert the process of iteration into a higher order function. For a set $\mathbb{T}$ let

$$I_{\mathbb{T}} : \mathbb{N} \longrightarrow \mathbb{T}'$$

be the function given by

$$I_{\mathbb{T}} xht = h^x t$$

for each $x \in \mathbb{N}, h \in \mathbb{T}', t \in \mathbb{T}$. Then

$$Ayx = I_{\mathbb{N}} xSy \quad Myx = I_{\mathbb{N}} x(Ay)0 \quad Eyx = I_{\mathbb{N}} x(My)1 \quad Bzyx = I_{\mathbb{N}} x(Ey)z$$

are rephrasings of the previous definitions. The measure of complexity is the nesting depth of the uses of $I_{\mathbb{N}}$.

Of course, there may be other, and better, measure of complexity, but at least this is a start.

What about about functions of complexity 5 and above?

Before we answer that let's take a closer look at B. This function is a version of the stacking function, which is an important gadget since it lives at the extremes in two different senses. It lies at the limit of those functions which can be described using commonly accepted notation, and it is as complex as any function we would normally meet.

3.4 DEFINITION. Let $\beth$ be the 3-placed function over $\mathbb{N}$ generated by

$$\beth(0, y, z) = z \qquad \beth(x', y, z) = y^{\beth(x,y,z)} = \beth(x, y, y^z)$$

for each $x, y, z \in \mathbb{N}$. There are alternative recursive steps, a head versions and a tail version. ■

We find that

$$Bzyx = \beth(x, y, z)$$

for all $x, y, z \in \mathbb{N}$. Pictorially we have

$$\beth(x, y, z) = y^{y^{y^{\cdot^{\cdot^{\cdot^{y^z}}}}}}$$

where there is a stack of x number of ys topped off with z. This picture shows why B and $\beth$ lie at the extremes of the commonly used notations. Try to find a pictorial form of the value

$$(Bzy)^x 0$$

obtained by suitably iterating the stacking process.

It takes a bit longer to explain why $\beth$ lies at the limits of the commonly met complexities, but let's make a start.

We need a cleaner, and more extensive version of the informal hierarchy.

3.5 EXAMPLE. For each $f : \mathbb{N}'$ let f^+ be given by

$$f^+ x = f^x 1$$

the xth-iterate of f evaluated at 1. By repeating this we obtain a sequence of functions. Thus we set

$$f^{[0]} = f \qquad f^{[r+1]} = f^{[r]+}$$

for each $r \in \mathbb{N}$.

Now consider the doubling function $\mathit{2}$ given by

$$\mathit{2}x = 2 \times x$$

for $x \in \mathbb{N}$. We find that

$$\mathit{2}x = 2 \times x \qquad \mathit{2}^+ x = 2^x \qquad \mathit{2}^{[2]} x = \beth(x, 2, 1)$$

and then $\mathit{2}^{[3]}, \mathit{2}^{[4]}, \mathit{2}^{[5]}, \ldots$ become wilder and wilder. ■

It is clear that in some sense the chain of 1-placed functions

$$(\mathit{2}^{[r]} \,|\, r < \omega)$$

is getting more and more complicated at each step, and the number of uses of $(\cdot)^+$ gives a measure of this complexity. We can use these functions as benchmarks against which we can compare other, more intricate, functions. In this way we produce upper bounds on the complexity of functions as measured by the $\mathit{2}^{[\cdot]}$ hierarchy.

The construction $(\cdot)^+$ is a higher order function $J \in \mathbb{N}''$, that is

$$f^+ = Jf$$

for each $f \in \mathbb{N}$. It is an example of a jump operator (for it is used to jump up the complexity of a function). There are several other commonly use jump operators and we will meet some of these in section 6. Also, we may vary the base function $\mathit{2}$, and all these possibilities give a quite extension collection of measuring techniques.

In the next section we look at a class of functions, the primitive recursive functions, which have played an important role in the development of this subject. Each of these functions has a 'finite complexity' but there is no single bound for the whole class. After that we will look at what happens below this complexity, and what happens above.

Exercises

14 Consider the two functions $\beth_h, \beth_t$ obtained by head and tail recursions in Definition 3.4.

(a) Show formally that these are the same function.

(b) Show that

$$\beth(l, y, \beth(m, y, z)) = \beth(l + m, y, z)$$

for each $l, m, y, z \in \mathbb{N}$.

15 Consider the iterator $I : \mathbb{N} \longrightarrow \mathbb{T}''$ over a set $\mathbb{T}$.

(a) Write down recursive specification of I, and classify it as head, body, or tail.

(b) Use the specification to show

$$Ixh \circ Iyh = I(y + x)h \qquad Ix \circ Iy = I(y \times x)h$$

for $x, y \in \mathbb{N}$ and $h \in \mathbb{T}'$.

(c) Can you find a compound which evaluates to $I(y^x)$.

16 Consider the algorithms A_l, A_r of Definition 3.2. In the two cases, for $x, y \in \mathbb{N}$ let $a(x, y)$ be the number of steps taking to reduces $A(\ulcorner x \urcorner, \ulcorner y \urcorner$ to $\ulcorner y + x \urcorner$ (that is, the cost of running the algorithm).

Write down a recursive specification of the two functions a. Show that the two functions are the same. Give an explicit description of this function.

17 Consider the algorithms M_l, M_r of Definition 3.2. In the two cases, for $x, y \in \mathbb{N}$ let $m(x,y)$ be the number of steps taking to reduces $M(\ulcorner x \urcorner, \ulcorner y \urcorner$ to $\ulcorner y \times x \urcorner$ (that is, the cost of running the algorithm). This, of course, may depend on which algorithm A is invoked.

Write down a recursive specification of each functions m. Give an explicit description of the function. Show that, in general. one algorithm is quadratically faster than the other. Can you suggest what causes this?

18 Neither of the two algorithms M_l, M_r of Definition 3.2 are tail recursive. Design a tail recursive algorithm for multiplication and estimate its cost.

19 Look up a proof of a formal verification in PA of the commutativity and associativity of addition (say in [6] pp 102–106 [*1964 edition*]. What is the crucial trick in the proof?

4 The primitive recursive functions

In section 3 we said that the stacking function $\beth$ lies at the limit in terms of the commonly used notations and complexities most often met. In this section we will see some functions of much greater complexity. We are going to look at a class of functions you should have seen before, the primitive recursive functions. However, when it come to being complex, these functions haven't even started, and in later section we will go way beyond them. This puts $\beth$ in its rightful place in the grand scheme of things.

There are some small and perfectly formed examples of first order function which are often ignored. For each pair $1 \leq i \leq s$ the projection function

$$\begin{array}{rcl} \mathbb{N}^s & \xrightarrow{\binom{s}{i}} & \mathbb{N} \\ \mathsf{x} & \longmapsto & x_i \end{array}$$

merely selects one of its inputs as indicated by its name. Here

$$\mathsf{x} = (x_1, \ldots, x_s)$$

is the tuple of inputs. These are useful little gadgets. For instance, if you want to shuffle around the order of the inputs of a function, or introduce some dummy inputs, then this can be done by composing with projections functions.

There is also a useful notion which can make several definition more compact. (This word has been used in parts of mathematics long before Dolly was even thought of.)

4.1 DEFINITION. A clone is a family of first order functions over $\mathbb{N}$ which contains all projections and is closed under composition. ■

For example, the family of all projection functions is the smallest clone. We look at more interesting examples.

The intersection of any family of clones is itself a clone. This enables us to make sense of 'the smallest clone such that ... ' where some closure condition is given. Here is the first use of this idea.

4.2 DEFINITION. An instance of primitive recursion is an instance of head recursion where the target type $\mathbb{T}$ is $\mathbb{N}$ and the parameter type $\mathbb{P}$ is $\mathbb{N}^r$ for some $r \in \mathbb{N}$. Thus the instance produces a $(1+r)$-placed function

$$\mathbb{N}^{1+r} \xrightarrow{\ f\ } \mathbb{N}$$

where conventionally we have placed the recursion argument in the left-most position.

The initial primitive recursive functions are the two functions

$$\begin{array}{ccc} \mathbb{N} & \xrightarrow{\ S\ } & \mathbb{N} \\ x & \longmapsto & 1+x \end{array} \qquad \begin{array}{ccc} \mathbb{N} & \xrightarrow{\ Z\ } & \mathbb{N} \\ x & \longmapsto & 0 \end{array}$$

and the projections.

The class $\mathcal{P}$ of primitive recursive functions is the smallest clone that contains the initial functions and is closed under primitive recursion.

More generally, for any class $\mathcal{G}$ of functions, the class $\boldsymbol{P}(\mathcal{G})$ of functions that are primitive recursive in $\mathcal{G}$ is the smallest clone that includes $\mathcal{G}$, contains the initial functions, and is closed under primitive recursion.

In particular, $\mathcal{P} = \boldsymbol{P}(\emptyset)$. ■

This is typical of the way many clones are produced. Suppose we have certain 1-step constructions which convert functions into functions. We may then consider the smallest clone which contains certain nominated initial functions and is closed under these 1-step constructions. A typical member of this clone is built up in a finite number of steps from the initial functions where at each step we use either a composition or one of the 1-step constructions.

In particular a primitive recursive function $g \in \mathcal{P}$ is built up from the initial functions S and Z by repeated use of composition and primitive recursion. That is, there is a list

$$g_0, g_1, g_2, \ldots, g_l = g$$

of functions where each g_i is either an initial function or is obtained from earlier functions by composition or an instance of primitive recursion. The length l if this list is a crude measure of the complexity of the constructed function g. (Strictly speaking, this is a measure of the particular construction used to produce g, not of g itself. There may be other constructions of g which are 'better' than this one.) The functions S, A, M, E, B are all members of $\mathcal{P}$, and have crude complexity 0,1,2,3,4, respectively.

We relativize this idea to obtain $g \in \boldsymbol{P}(\mathcal{G})$ by merely adding $\mathcal{G}$ to the stock of initial functions.

At this stage a first introduction to primitive recursion would develop a whole batch of examples, some to illustrate how the recursion works, some because they are needed later, and some to show that the class $\mathcal{P}$ contains some surprising functions. For instance, in [6] Mendelson devotes some 10 pages [***in the 1966 edition***] to this kind of material. A more leisurely account is given in [1]. A comprehensive account of this and related material can be found in [9] (This book was first written by a Hungarian in German in 1950. It was translated into English in 1965, and that edition contains some substantial additions. It is still the most detailed, and on some points the only, account of much of this material.) If you look at Gödel's original incompleteness paper, which is translated in the collection [2], you will find a list of 46 small examples each showing that a certain function is primitive recursive. Some third rate philosophers have had the idea that the fine details of these examples are the very essence of incompleteness.

We won't go through a selection of primitive recursive functions (because we don't need them) but it is worth giving a couple of the more striking examples.

4.3 EXAMPLE. The function

$$i \longmapsto p_i$$

which enumerates the prime numbers in ascending order is primitive recursive.

The 2-placed function

$$\exp(n, i)$$

which returns the exponent of p_i in the prime factorization of n is primitive recursive. ■

It is not at all clear how either of these functions can be built up in a primitive recursive fashion. If you haven't see it done, you should try to work out a method. Even if you have seen it done, you should try to work through the method. At some point a construction of the first function will need some non-trivial mathematics (such as a primitive recursive bound on the ith prime).

Here is perhaps a more striking example.

4.4 PROPOSITION. *The 2-placed function*

$$\text{append} : \mathbb{N}^2 \longrightarrow \mathbb{N}$$

given by

$$\text{append}(a, x) = 2^a \times \prod_i p_{i+1}^{e(i)} \quad \textit{where} \quad x = \prod_i p_i^{e(i)}$$

is primitive recursive (and in fact Kalmár elementary).

Functions such as these are most often used to code a list of natural numbers as a natural number. Thus the list

$$[a(0), a(1), \dots, a(l)]$$

is coded as

$$a = p_0^{a(0)} p_1^{a(1)} \cdots p_l^{a(l)}$$

and the functions enable us to move between the list and its code, and to manipulate the coded lists. Almost every presentation of the properties of recursive functions uses a lot of coding. This is a bad idea, for it tends to obscure what is going on. If you work out what the append function is doing in terms of lists then you will see that in the uncoded world this operation ought to have zero complexity. However, the coding gives it unnecessary pretensions.

(Actually, there is an aspect of this that is being deliberately overlooked, but that doesn't detract from the point being made.)

When we look at While loops you will see that we can not use coding, and we have to analyse the actual situation, not some encrypted version of it.

The clone $\mathcal{P}$ contains some obvious functions and some surprising ones. How complicated can members of $\mathcal{P}$ become? To gain some insight we try to get to the extremes of this clone. We have seen already a way of doing this.

4.5 EXAMPLE. Consider the jump operator $(\cdot)^+$ of Example 3.5. Thus

$$f^+x = f^x 1$$

for each $f \in \mathbb{N}', x \in \mathbb{N}$. A simple calculation shows that f^+ is primitive recursive in f, that is $f^+ \in \boldsymbol{P}(f)$. In particular, each member of the generated sequence

$$(f^{[r]} \mid r < \omega)$$

is primitive recursive in f. ■

It turns out that for most reasonable functions f the generated sequence 'cofinally exhausts' $\boldsymbol{P}(f)$, and the 'diagonal limit'

$$x \longmapsto f^{[x]}1$$

is not in $\boldsymbol{P}(f)$. We won't be able to develop these two ideas in any great detail, but we can give a decent impression of what they mean.

We need a method of comparing an arbitrary function (which may be many-placed) against a 1-placed function we are using as a benchmark. We do this using rate of growth. This is analogous to comparing the derivatives of two real valued functions, except here we don't have a notion of derivative.

We need some restrictions on the benchmark functions.

4.6 DEFINITION. A 1-placed function $f \in \mathbb{N}'$ is, respectively,

inflationary strictly inflationary

if

$$x \leq fx \qquad x < fx$$

holds for each $x \in \mathbb{N}$.

A 1-placed function $f \in \mathbb{N}'$ is monotone if

$$x \leq y \Longrightarrow fx \leq fy$$

holds for all $x, y \in \mathbb{N}$.

A bounding function is a 1-placed function that is both strictly inflationary and monotone. ■

Two monotone functions f, g can be compared pointwise, that is

$$g \leq f \Longleftrightarrow (\forall x)[gx \leq fx]$$

and this gives a partial ordering of such functions. More generally, for $a \in \mathbb{N}$ we let

$$g \leq_a f \Longleftrightarrow (\forall x \geq a)[gx \leq fx]$$

to give an 'eventual comparison' of g and f. We use a further extension of this idea to compare an arbitrary many-placed function with a bounding function.

Here is the appropriate notion in all its glory. You need not worry too much about this, for we won't use it any technical work. However, in a more detailed and comprehensive account it has a central role to play. There is an explanation of the notion right after the formal definition.

4.7 DEFINITION. Let $f \in \mathbb{N}'$ be a bounding function.

For an s-placed function $g : \mathbb{N}^s \longrightarrow \mathbb{N}$ we write

$$g \sqsubseteq f$$

and say g is eventually dominated by f if there is some $a \in \mathbb{N}$ such that

$$a, x_1, \ldots, x_s \leq x \Longrightarrow g(x_1, \ldots, x_s) \leq fx$$

holds for all $x_1, \ldots, x_s, x \in \mathbb{N}$. ■

This comparison can be described in a different way. Given an s-placed function g let

$$\overline{g}x = \max\{x + 1, g(x_1, \ldots, x_s) \mid x_1, \ldots, x_s \leq x\}$$

for each $x \in \mathbb{N}$ to obtain a 1-placed function $\overline{g}$. It is easy to check that $\overline{g}$ is strictly inflationary and monotone. Then

$$g \sqsubseteq f \iff (\exists a)[\overline{g} \leq_a f]$$

is an equivalent way of defining eventual domination.

We are not going to do any manipulations with these notions, but there are two aspects that should be explained. We impose the two conditions on the bounding function f (strictly inflationary and monotone) to make the comparisons between functions easier to handle. There is a similar reason for the use of the 'start' input a. In general we don't care what happens to a function g in some finite region near to the origin.

It is easy to check that if f is a bounding function then so is f^+. In particular, each function $\mathcal{2}^{[r]}$ is a bounding function. With a bit of effort we can prove the following.

4.8 PROPOSITION. *Each primitive recursive function g is eventually dominated by $\mathcal{2}^{[r]}$ for some r (depending in g).*

We will not prove this result (or any result given as a Proposition). It is not difficult to prove, but does take a bit of time, and we are not going to develop the methods used.

What is important is that the result gives us a measure of the complexity of a primitive recursive function which agrees with our intuitive notion of complexity.

Let's first look at an idea that doesn't quite work.

For each $r < \omega$ let $\mathcal{Q}_r$ be the class of primitive recursive functions g that are eventually dominated by $\mathcal{2}^{[r]}$. Thus

$$\mathcal{Q}_0 \subseteq \mathcal{Q}_1 \subseteq \mathcal{Q}_2 \subseteq \mathcal{Q}_3 \subseteq \cdots$$

and $\mathcal{P}$ is just the union of this ω-chain. Put another way, for each $g \in \mathcal{P}$ there is a smallest $r < \omega$ with $g \in \mathcal{Q}_r$, and we can think of this r as a measure of the complexity of g. Unfortunately, this is not quite the right idea.

Notice the these families $\mathcal{Q}_r$ are not clones, for they are not closed under composition. We can correct this by closing off under composition. However, there is also a more serious flaw.

Think of the construction

$$g_0, g_1, g_2, \ldots, g_l = g$$

of a function $g \in \mathcal{P}$. It could be that $g = g_l$ is eventually dominated by some function but some of the components g_i are not. For instance, the numerical negation function $\mathrm{Neg} \in \mathbb{N}'$ given by

$$\mathrm{Neg}0 = 1 \quad \mathrm{Neg}x' = 0$$

is primitive recursive, and hence so is $\text{Neg} \circ g$ for each primitive recursive function g. This composite function ought to be as least as complicated as g, but it is dominated by $\mathit{2}$ whereas g may not be.

A better measure of complexity is achieved if we bound all of the components in the construction of g.

4.9 DEFINITION. For each $r < \omega$, a primitive recursive function g with construction

$$g_0, g_1, g_2, \ldots, g_l = g$$

is a member of the class $\mathcal{P}_r$ if there is some $s \in \mathbb{N}$ such that

$$g_i \sqsubseteq (\mathit{2}^{[r]})^s$$

for each $i \leq l$. ■

There are two modifications here to the previous idea. Firstly we use some finite iterate of a function to do the bounding. This ensures that the class obtained is a clone. Secondly, we bound each of the functions occurring in the construction, not just the final outcome. In other words we choose a clone inside which the complete construction can be carried out.

There is a subtlety here which we have already hinted at and should be mentioned again. After that we can be ignore it.

It is possible to have two different constructions

$$g_0, g_1, g_2, \ldots, g_l = g \qquad h_0, h_1, h_2, \ldots, h_l = h$$

of the same function $g = h$. Which should be used to measure the complexity? In fact, it isn't the function that carries the complexity, but the construction, and the algorithm embedded in the construction. When we convert such a construction into a program (for evaluating the end function) this distinction becomes clearer. It is common, but bad, practice not to distinguish between a function and its construction, and we will conform with that here (merely because we aren't going to take this material far enough for it to matter).

Any proof of Proposition 4.8 can be reworked to give the following.

4.10 PROPOSITION. *The construction of Definition 4.9 gives a strictly ascending chain*

$$\mathcal{P}_0 \subseteq \mathcal{P}_1 \subseteq \mathcal{P}_2 \subseteq \mathcal{P}_3 \subseteq \cdots$$

of clones with union $\mathcal{P}$.

A result such as this suggests several questions.

For the purposes of this course we accept the importance of the clone $\mathcal{P}$ of primitive recursive functions. What about the subclones $\mathcal{P}_r$, do these have any importance? Since

$$\mathit{2}^{[0]}x = \mathit{2}x = 2x$$

we have

$$(\mathit{2}^{[0]})^s x = 2^s x$$

and hence $\mathcal{P}_0$ is the clone of functions whose entire construction is bounded by $x \mapsto mx$ for some $m \in \mathbb{N}$. These are not very interesting functions (when constructed in this fashion). In fact, the class $\mathcal{P}_0$ is included in the chain merely to make the subscripting a little neater. With a bit of reorganization it could be removed. The class $\mathcal{P}_1$ is important in its own right, and we look at this in the next section. In a way it forms the frontier between two distinct areas and ways of doing things.

How sensitive is this hierarchy to the base function $\mathit{2}$ and the jump operator $(\cdot)^+$ used to generate it? Both of these can be modified to produce other layerings of $\mathcal{P}$, or different ways of generating the same layering. For instance, we could start with either of

$$fx = 2 + x \qquad fx = 2^x$$

to produce essentially the same results. Later we will look at some other jump operators which, in some respects, are more convenient.

Is there anything beyond this hierarchy? Put another way, are there methods of jumping right out of the hierarchy, and if so just how far can we jump? We consider this question in section 6.

In the literature you will find a chain

$$\mathcal{E}_0 \subseteq \mathcal{E}_1 \subseteq \mathcal{E}_2 \subseteq \mathcal{E}_3 \subseteq \cdots$$

of clones with union $\mathcal{P}$. It turns out that

$$\mathcal{E}_{r+2} = \mathcal{P}_{r+1}$$

for all r. That is $\mathcal{E}_3 = \mathcal{P}_1$ and there after the two chains agree. The lower layers $\mathcal{E}_0, \mathcal{E}_1, \mathcal{E}_2$ are included for historical reasons (and correspond approximately to the clones of functions which, in some sense, are like S, A, M, respectively).

Exercises

20 Consider the layering of $\mathcal{P}$ generated using the operator $(\cdot)^+$ but starting from the base function

$$(a) \quad fx = 2 + x \qquad (m) \quad gx = 2 \times x \qquad (e) \quad hx = 2^x$$

in turn. (So case (m) is the one done in the section.) Show that these three base functions produce essentially the same layering. That is, after a couple of steps the same clones are generated but perhaps with a different index.

21 Let f be an arbitrary 1-placed function f.

Show that the jumped up version f^+ is primitive recursive in f.

Show that if f is a bounding function then so is f^+.

By iteration $(\cdot)^+$ we obtain a chain $(f^{[r]} \mid r < \omega)$ of functions each of which is primitive recursive in f. Does this mean that the 2-placed function

$$(r, x) \longmapsto f^{[r]}x$$

is primitive recursive in f?

22 Can you suggest a function in $\mathcal{P}_{r+1} - \mathcal{P}_r$ (for each $r < \omega$)?

Can you suggest a function that is recursive in some sense but is not primitive recursive?

23 Show that for each bounding function f and arbitrary function g we have

$$g \sqsubseteq f \Longrightarrow (\exists r)[\overline{g} \leq f^r]$$

where $\overline{g}$ is the bounding function canonically asssociated with g.

24 As a long term project try to work out a proof of Proposition 4.4.

25 Consider the function

$$P : \mathbb{N}^2 \longrightarrow \mathbb{N}$$

given by

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

for $x, y \in \mathbb{N}$.

(a) Check that this value is a natural number and describe a 'natural' way of arriving at it.

(b) Show that P sets up a bijection between $\mathbb{N}^2$ and $\mathbb{N}$.

(c) Describe two functions $L, R \in \mathbb{N}'$ such that

$$L(P(x, y)) = x \qquad R(P(x, y)) = y \qquad P(Lz, Rz) = z$$

for all $x, y, z \in \mathbb{N}$.

(d) Show that P, L, R are primitive recursive (in fact, Kalmár elementary).

Any function P, L, R satisfying the first two identities form a triple of pairing gadgets. If the third identity holds then they have surjective pairing. This particular example was devised by Cantor.

(e) Can you suggest 'easier' functions that form pairing gadgets (perhaps without surjective pairing)?

> Many recursions can be rephrased as iterations, provided we used higher order gadgets and certain other tricks. The first exercise gives a method that is quite general, and the second exercise gives a method that is more specific. You should compare these two methods.

26 Consider a function

$$\mathbb{N}, \mathbb{P} \xrightarrow{f} \mathbb{T}$$

specified by a b/s body recursion using the usual data functions g, h, k. Let

$$\mathbb{G} = (\mathbb{P} \longrightarrow \mathbb{T})$$

so that

$$f : \mathbb{N} \longrightarrow \mathbb{G}$$

when viewed in curried form. Using the data functions h, k consider

$$H : \mathbb{N} \times \mathbb{G} \longrightarrow \mathbb{N} \times \mathbb{G}$$

given by

$$H(x, \phi) = (x', \psi) \qquad \text{where } \psi p = h(x, p, \phi p^+) \text{ for } p^+ = k(x, p)$$

for $x \in \mathbb{N}, \phi \in \mathbb{G}, p \in \mathbb{P}$.

Show that

$$H(x, fx) = (x', fx')$$

and hence

$$H^x(0, g) = (x, fx)$$

for each $x \in \mathbb{N}$.

27 Consider a function

$$\mathbb{N}, \mathbb{P} \xrightarrow{f} \mathbb{T}$$

specified by a b/s head recursion using the usual data functions g, h. Let

$$H : \mathbb{P} \longrightarrow (\mathbb{N} \times \mathbb{T})'$$

be given by

$$Hp(x, t) = (x', h(x, p, t))$$

for $p \in \mathbb{P}, x \in \mathbb{N}, t \in \mathbb{T}$.

Show that

$$(Hp)^x(0, gp) = (x, f(x, p))$$

for $p \in \mathbb{P}, x \in \mathbb{N}$.

Can you extend this method to deal with body recursion?

5 Deep inside primitive recursion

The received wisdom is that the class $\mathcal{P}$ of primitive recursive functions has a distinguished place within the class of all 'computable' or 'recursively specifiable' functions. Certainly, it is the non-trivial clone with the simplest definition, and the one most often seen in any account of recursion. (However, as with much of received wisdom, there seems to be little justification for believing that the clone $\mathcal{P}$ is anything very special.)

We have seen that $\mathcal{P}$ contain some pretty wild functions, as measured by speed, that is rate of growth. More generally, for each bounding function f the clone $\boldsymbol{P}(f)$ of those functions primitive recursive in f contains functions which are miles a head of f in terms of speed. In the next section we will put a value on this jump in speed.

This observation suggests that we look for a subclone of $\mathcal{P}$ which isn't so wild but does contain all of the functions we might need in most day to day mathematics. In fact, the class $\mathcal{P}_1$ is just such a clone. In this section we describe this clone in a quite different way.

We will generate this clone, which we call $\mathcal{K}$, by the standard method. Thus it is the smallest clone which contains certain initial functions and closed under certain restricted form of primitive recursion.

The function in $\mathcal{K}$ are often described as the 'elementary' functions. Here the word is not used in the sense of 'first order' but in the sense of 'everyday, non-exotic', just as an 'elementary proof' of the prime number theorem, say, would not use any complex analysis. In other words 'elementary' here means built out of addition, multiplication, and the like.

The relevant restricted forms of primitive recursion have this nature.

5.1 DEFINITION. Each function

$$\mathbb{N}, \mathbb{P} \xrightarrow{\ k\ } \mathbb{N}$$

generates a function f of the same type by

bounded sum	bounded product
$f(0,p) = 0$	$f(0,p) = 1$
$f(x',p) = f(x,p) + k(x,p)$	$f(x',p) = f(x,p) \times k(x,p)$

respectively, if the indicated equalities hold for each $x \in \mathbb{N}$ and $p \in \mathbb{P}$. ■

Clearly both these constructions are instances of head recursion, and when $\mathbb{P}$ is a power of $\mathbb{N}$, then it they are instances of primitive recursion. A more common notation for the functions is something like

$$f(x,p) = \sum_{i<x} k(i,p) \qquad f(x,p) = \prod_{i<x} k(i,p)$$

which indicates where the names come from. We use these constructions for first order functions k, that is where $\mathbb{P} = \mathbb{N}^s$ for some $s \in \mathbb{N}$.

As with $\mathcal{P}$ we use a small collection of initial functions, except for $\mathcal{K}$ we need slightly more of them. As with $\mathcal{P}$ we use the projections and Z, S, but now we throw in A, M and a 2-placed decision function. This is usually taken to be stunted subtraction given by

$$D(x, y) = \begin{cases} 0 & \text{if } y \le x \\ y - x & \text{if } x \le y \end{cases}$$

(for $x, y \in \mathbb{N}$).

5.2 DEFINITION. The class $\mathcal{K}$ of Kalmár elementary functions is the least clone which contains certain initial functions and is closed under bounded sums and bounded products (for first order functions).

More generally, for any class $\mathcal{G}$ of functions, the class $\boldsymbol{K}(\mathcal{G})$ of functions that are Kalmár elementary in $\mathcal{G}$ is the smallest clone that includes $\mathcal{G}$, contains the initial functions, and is closed under bounded sums and bounded products (for first order functions).

In particular, $\mathcal{K} = \boldsymbol{K}(\emptyset)$. ■

As indicated above the word ‘elementary’ here is not a reference to ‘first order’ but to the fact that these functions are rather simple, and are the kind of function you find in everyday mathematics. The class was first isolated by L. Kalmár and his name is sometime used to distinguish this notion of elementary from others.

Almost all the functions met in everyday mathematics are in $\mathcal{K}$. For instance, using the projection function k with $k(x, y) = y$ we have

$$\sum_{i<x} k(i, y) = y \times x \qquad \prod_{i<x} k(i, y) = y^x$$

so that the exponentiation function E is in $\mathcal{K}$. Some surprising functions are in $\mathcal{K}$. For instance, the function

$$x \longmapsto p_x$$

which enumerates the primes is in $\mathcal{K}$. A proof of this is outlined in the exercises.

Since exponentiation is in $\mathcal{K}$ we see that for each $r \in \mathbb{N}$ the 2-placed function $\beth(r, \cdot, \cdot)$ is in $\mathcal{K}$. These stunted stacks provide a layering of $\mathcal{K}$, as the following analogue of Proposition 4.8 shows.

5.3 PROPOSITION. *Each member g of $\mathcal{K}$ is eventually dominated by $\beth(r, 2, \cdot)$ for some r (depending on g).*

The proof of this uses the same general method as that of Proposition 4.8 but is easier. It makes use of certain comparisons which mix nicely with the generating constructions. An idea of this proof is given in the exercises.

Using this result we may decompose $\mathcal{K}$ as the union of an ascending ω-chain

$$\mathcal{K}_0 \subseteq \mathcal{K}_1 \subseteq \mathcal{K}_2 \subseteq \cdots$$

of subclasses. As with the previous layerings, the class $\mathcal{K}_0$ has no interests, but even the class $\mathcal{K}_1$ (where everything done is bounded by 2^x is quite complicated).

Notice that these classes $\mathcal{K}_r$ are not clones. When we get inside $\mathcal{K}$ the appropriate measures of complexity are stricter than those used higher up. At this level a composite of two functions may not be considered a trivial operation. As an example, for some purposes a quadratic polynomial bound can be an acceptable complexity whereas one of degree 17 is not. We say a bit more about this at the end of this section.

How are the two classes $\mathcal{K}$ and $\mathcal{P}$ related? Trivially we have $\mathcal{K} \subseteq \mathcal{P}$, but which part of $\mathcal{P}$ does $\mathcal{K}$ give?

5.4 PROPOSITION. *We have $\mathcal{K} = \mathcal{P}_1$, that is the Kalmár elementary functions form the first interesting layer of the class of primitive recursive functions.*

More generally, we have $\boldsymbol{K}(\mathcal{P}_r) = \mathcal{P}_r$ for each $0 \neq r \in \mathbb{N}$.

We have defined the clone $\mathcal{K}$ and the operator $\boldsymbol{K}$ in terms of weak recursion properties. We did this to get a direct comparison with the clone $\mathcal{P}$ and the operator $\boldsymbol{P}$. There is another way of getting at $\mathcal{K}$ and $\boldsymbol{K}$ which in some ways is more enlightening. This uses a search operator to produce new functions. Strictly speaking this is not a recursion in the sense of these notes, but it is often classified as one (since it is always used in conjunction with recursion).

5.5 DEFINITION. For a given data function h, as to the left,

$$\mathbb{N}, \mathbb{P} \xrightarrow{\quad h \quad} \mathbb{N} \qquad\qquad \mathbb{N}, \mathbb{P} \xrightarrow{\quad f \quad} \mathbb{N}$$

let f, with the same type, be the function given by

$$f(x,p) = \begin{cases} \text{the least } y \leq x \text{ with } h(y,p) = 0 & \text{if there is such a } y \\ x+1 & \text{if there isn't} \end{cases}$$

for $x \in \mathbb{N}, p \in \mathbb{P}$. We say f is obtained from h by bounded search. ■

There is a common notation to describe this construction. We write

$$f(x,p) = (\mu y \leq x)[h(y,p) = 0]$$

where '$\mu y \leq x$' is read 'the least $y \leq x$ such that ...' with the cut-off given by the input x. To calculate $f(x,p)$ we first evaluate

$$h(0,p),\ h(1,p),\ \ldots,\ h(y,p),\ \ldots,\ h(x,p)$$

in turn, and see if any of these values is 0. If there is such a value, then we take the first witness y. Otherwise we take $x+1$. Thus, from the value of $f(x,p)$ we can tell whether or not this search is successful, and if it is, where the first success occurred.

We used bounded search for functions

$$h : \mathbb{N} \times \mathbb{N}^s \longrightarrow \mathbb{N}$$

that is where $\mathbb{P} = \mathbb{N}^s$.

5.6 PROPOSITION. *For each $(1+s)$-placed function h the function f given by*

$$f(x,p) = (\mu y \leq x)[h(y,p) = 0]$$

is in $\boldsymbol{K}(h)$.

Again a proof of this is not difficult. It consists of various tricks to massage the bounded search operator into a use of bounded sums and products. In a longer course it can easily be included (because there is time to set up the relevant tricks).

The crucial aspect of bounded search is that a nominated input x of the constructed function f acts as a bound on the length of the search. We know at the outset how many values of the data function h we might need. Of course, we can try the same idea without this bound built in. We will return to this in section 7.

The class $\mathcal{K} = \mathcal{P}_1$ is a natural boundary between two areas of study.

Within $\mathcal{K}$ we find the study of the feasibility of functions. This is mainly concerned with the subclass $\mathcal{K}_1$ but can sometimes go beyond that. The investigations use the methods of complexity theory and, more recently, finite model theory (which has nothing whatsoever to do with model theory). This is a rather unstructured subject which tends to attract a certain kind a 'mathematician' with a limited appreciation of other parts of mathematics.

In the other direction we can produce larger and larger classes of functions all of which are effective in some obvious sense. The remainder of these notes gives a flavour of that topic.

Exercises

28 Show that the two decision functions given by

$$\mathrm{Neg}(x) = \begin{cases} 0 \text{ if } x \neq 0 \\ 1 \text{ if } x = 0 \end{cases} \qquad \mathrm{Eq}(x,y) = \begin{cases} 1 \text{ if } x \neq y \\ 0 \text{ if } x = y \end{cases}$$

(for $x, y \in \mathbb{N}$) are in $\mathcal{K}$ (and, in fact, very low down in $\mathcal{K}$).

29 We know that $\mathcal{K} = \mathcal{P}_1 \subseteq \mathcal{P}_2$. Find a function in $\mathcal{P}_2 - \mathcal{K}$.
Can you suggest a function that is in $\mathcal{K}_{r+1} - \mathcal{K}_r$ (for arbitrary r)?

30 This exercise shows that the primes are enumerated by a function in $\mathcal{K}$. The proof uses several auxiliary functions.

(a) Show that the decision function given by

$$P(z) = \mathrm{Neg}^2(z) \times \mathrm{Neg}(\mathrm{Eq}(z,1)) \times \prod_{y<z}\prod_{x<z} \mathrm{Neg}(\mathrm{Eq}(z,yx))$$

(for $z \in \mathbb{N}$) test for being a prime.

(b) For each $x \in \mathbb{N}$ let $\pi(x)$ be the nuber of primes $p < x$ (not $p \leq x$ as is more usual). Show that $\pi \in \mathcal{K}$.

(c) Using bounded search show that the function $x \longmapsto p_x$ is in $\mathcal{K}$.

31 This exercise gives a proof of Proposition 5.3.

The result is stated in terms of eventual domination but, by a use of Exercise 23 we can work with domination.

The proof is a series of comparisons between various constructed functions and a selected bounding function.

Let f be any bounding function with $2^{2^x} \leq fx$ for all $x \in \mathbb{N}$. This ensures that $x^2 \leq x^x \leq fx$ holds for all $x \in \mathbb{N}$.

(a) Consider a composite function

$$k = h \circ (g_1, \ldots, g_l)$$

where each component $g_1, l \ldots, g_l, h$ is dominated by f. Show that k is dominated by f^2.

(b) Show that if the functions k is dominated by f then the function

$$\sum_{i<x} k(i,p)$$

is dominated by f^2.

(c) Prove a similar result concerning bounded products.

(d) Show that each $g \in \mathcal{K}$ is dominated by $\beth(r,2,\cdot)$ for some r (depending on g).

32 This exercise gives a proof of Proposition 5.6.

Given the function h let

$$k(x,p) = \prod_{i<x} \mathrm{Neg}^2(h(i,p)) \qquad l(x,p) = \sum_{i<x} k(i',p) \qquad f(x,p) = l(x',p)$$

for $x \in \mathbb{N}$ and parameters p. Observe that $k, l, f \in \boldsymbol{K}(h)$.

Show that f is the required bounded search function.

6 Way beyond primitive recursion

In the previous sections we looked at the class $\mathcal{P}$ of primitive recursive functions and the much smaller class $\mathcal{K}$ of Kalmár elementary functions. We saw that $\mathcal{P}$ could be layered into clones of complexity with $\mathcal{K}$ as the smallest interesting layer. In fact, most (if not all) of the common place functions live in $\mathcal{K}$, and it takes a little bit of work to jump out of this clone. The stacking function $\beth$ achieves this jump.

What about the clone $\mathcal{P}$? How can we jump out of this?

It is trivial to see that there must be first order functions (in fact members of $\mathbb{N}'$) outside $\mathcal{P}$. Each primitive recursive function has a construction which can be described by a finite piece of syntax. (The λ-calculus gives the syntactic methods for doing this.) In particular, the class $\mathcal{P}$ is countable. Since $\mathbb{N}'$ is uncountable, this means that most first order functions are not in $\mathcal{P}$.

This kind of argument gives us the mere existence of certain functions but tells us nothing about what these may look like, or how complex they may be.

A more interesting question is whether there are any functions outside $\mathcal{P}$ that can be generated by some form of recursion, and if so what kind of recursion is needed.

By definition, primitive recursion will not take us outside $\mathcal{P}$. It turns out that minor tweaks to primitive recursion are not powerful enough. A first introduction to these matters often gives several examples of this. Let's mention briefly some of the variations which stay within $\mathcal{P}$.

6.1 EXAMPLE. The first variant is course of values recursion. This is like primitive recursion except the recursion step makes use of several earlier values. Thus the fibonaci function is the best known example of this. A simple coding trick reduces this to primitive recursion. Suppose

$$f : \mathbb{N} \times \mathbb{P} \longrightarrow \mathbb{N}$$

is the constructed function (where $\mathbb{P}$ will be $\mathbb{N}^s$ for some s). We look at the function

$$F : \mathbb{N} \times \mathbb{P} \longrightarrow \mathbb{N}$$

given by

$$F(x, p) = \prod_{i<x} p_i^{f(i,p)}$$

using the sequence p_i of primes. (Do not confuse a prime p_i with a parameter p.) Using the various prime-related functions the given recursion is easily translated into a primitive recursive construction of F. ■

This example is typical of the many reductions to primitive recursion. It depends on several coding tricks. Sometimes the tricks are fairly straight forward (once discovered), but sometimes they are not.

6.2 EXAMPLE. A more complicated variant is primitive recursion with variation of parameters. This is just a head recursion (of Table 1) where the data functions are first order. Thus $\mathbb{T} = \mathbb{N}$ and $\mathbb{P} = \mathbb{N}^s$ for some s. It is rare to find a proof of a reduction of this to primitive recursion. Perhaps this is because a rather more sophisticated coding trick is needed. ■

One of the problems with the usual proofs of these results is that they make too much use of coding tricks, and this tends to obscure what is going on. A non-coding view (which requires us to look beyond the natural numbers) often makes things clearer. Try the following.

6.3 EXAMPLE. Let h be any 2-placed function over $\mathbb{N}$ and consider

$$f : \mathbb{N}^2 \longrightarrow \mathbb{N}$$

specified by

$$f(0, y) = y \qquad f(x', y) = h(x, f(x, t)) \quad \text{where } t = f(x, y)$$

(for $x, y \in \mathbb{N}$). At first sight it is not even clear that there is such a function. However, once we stop thinking in terms of natural numbers it becomes a rather simple function. ■

Many other variants of recursion can be reduced to primitive recursion. Thus we may look at recursions over several variables at one, or perhaps a recursion producing several functions at once. In most cases, sometimes with a little effort cases, we obtain a version of the following result.

6.4 PROPOSITION. *Let fancy-rec be one of a whole list of constructions by recursion. Then any clone $\mathcal{C}$ which closed under primitive recursion is also closed under fancy-rec.*

The most comprehensive account of these results is in [9], where some 30 pages are devoted to the topic. However, this is not a gripping read, and gives little insight into why these results are true.

So how can we jump out of $\mathcal{P}$ without jumping too far?

The first example of a function that is not primitive recursive but is clearly recursive in some sense was given by Ackermann. Naturally, this function is known as 'Ackermann's function'. Unfortunately, the example usually given in the literature is not due to Ackermann, but is a purported simplification by others. Let's look at Ackermann's original function and some variants.

To jump out of $\mathcal{P}$ we need to use a combination of multi-recursions together with a nesting of the function within the specification.

6.5 DEFINITION. Given a single data function $f \in \mathbb{N}'$ we specify three multi-placed functions.

$$\begin{array}{lll} F(0,0,x) = fx & F(0,x) = fx & F(0,x) = fx \\ F(i',0,x) = F(i,x,x) & F(i',0) = F(i,1) & F(i',0) = 2 \\ F(i,r',x) = F(i,0,y) & F(i',x') = F(i,y) & F(i,x') = F(i,y) \\ \text{where } y = F(i,r,x) & \text{where } y = F(i',x) & \text{where } y = F(i',x) \end{array}$$

where $i, r, x \in \mathbb{N}$. Of course, there are three different functions F. ■

These three examples are not the same function (unless the base function f is rather trivial), but they all have roughly the same complexity. The left hand function is Ackermann's example (when the base function f is successor). The central function is the one most often seen in the literature.

How do we show that, for a reasonable base function f, this function jumps out of $\boldsymbol{P}(f)$? We compare certain rates of growth. Not the way it is usually done, but the way Hilbert suggested it should be done. We are not going to go through the details of this proof, but we will look at the way the general mechanisms are set up.

We extend the method used in section 4. There we started from a base function $f \in \mathbb{N}'$ of a certain kind and an operator $J \in \mathbb{N}''$ with certain properties. We used the examples $f = \mathit{2}$ and $J = (\cdot)^+$, but other examples will work equally well. Any bounding function (that is, strictly inflationary and monotone function) can be used as the base. The most common choice is successor.

Given such a base function $f \in \mathbb{N}$ and a suitable operator $J \in \mathbb{N}''$ we set

$$f^{[i]} = J^i f$$

for each $i < \omega$ to produce an ω-chain of functions which. When f is a rather trivial bounding function these $f^{[i]}$ are the milestones which indicate the boundaries of the layers of $\mathcal{P}$. More generally, they indicate a layering of $\boldsymbol{P}(f)$.

To jump out of $\mathcal{P}$ it is sufficient to dominate each of $f^{[i]}$, and there is an obvious way to do this.

For various reasons it turns out that the operator $(\cdot)^+$ is not as amenable as it first appears. We modify it slightly.

6.6 DEFINITION. The three higher order functions

$$A, R, B : \mathbb{N}' \longrightarrow \mathbb{N}'$$

are given by

$$Afx = f^{x+1}x \qquad Rfx = f^{x+1}1 \qquad Bfx = f^x 2$$

for each $f \in \mathbb{N}'$ and $x \in \mathbb{N}$.

Each of A, R, B is an example of a jump operator as in the operator $(\cdot)^+$ used earlier. These are used to jump up the rate of growth of an inflationary and monotone function.

We give only a partial proof of the following result. The remainder of the proof is left as an exercise. The induction techniques used in the proof are important.

6.7 THEOREM. *For the functions F specified in Examples 6.5 we have, respectively*

$$F(i, r, x) = (A^i f)^{r+1} x \qquad F(i, x) = R^i f x \qquad F(i, x) = B^i f x$$

for each $i, r, x \in \mathbb{N}$.

Proof. For a complete proof of each part we should verify several things.

- There is at most one function satisfying the specification.
- There is at least one function satisfying the specification.
- The unique solution is total.
- The unique solution is that given above.

Of course, it is possible to combine some of these verifications. For instance, it is quite easy to check in each case that the function given above does satisfy the specification. In fact this follows by a minor modification of what we do shortly. We will show that if any function meets the specification, then that function must be the one given above.

We fill in the details for the left-hand F of Definition 6.5.

We proceed by a double induction over the recursion variables i, r. To help with this let

$$\begin{array}{ll} [i, r] & \text{abbreviate } (\forall x : \mathbb{N})[F(i, r, x) = (A^i f)^{r+1} x] \\ [i, \cdot] & \text{abbreviate } (\forall r : \mathbb{N})[i, r] \\ [\cdot, r] & \text{abbreviate } (\forall i : \mathbb{N})[i, r] \\ [\cdot, \cdot] & \text{abbreviate } (\forall i, r : \mathbb{N})[i, r] \end{array}$$

so that $[\cdot, \cdot]$ is the eventually objective.

Using the three clauses of the specification we see that

$$(1) \quad [0, 0] \qquad (2) \quad [i, \cdot] \Longrightarrow [i', 0] \qquad (3) \quad [i, 0] \wedge [i, r] \Longrightarrow [i, r']$$

hold for each i, r. For instance, to verify (3), with

$$y = F(i, r, x)$$

we have

$$y = (A^i f)^{r+1} x$$

by $[i, r]$, and then

$$F(i, r', x) = F(i, 0, y) = A^i f y = (A^i f)((A^i f)^{r+1} x) = (A^i f)^{r+2} x$$

to give $[i, r']$. Here the first equality follows by the third clause of the specification, and the second follows by the hypothesis $[i, 0]$.

Now (3) gives

$$[i, 0] \Longrightarrow [i, \cdot]$$

by an induction over r, so that

$$[i, 0] \Longrightarrow [i', 0]$$

by (2), and hence

$$[\cdot, 0]$$

by an induction over i using (1). This with the first implication of this paragraph gives $[\cdot, \cdot]$, as required. ■

The other two parts of this result are proved in a similar way but, of course, each uses its own particular trick.

It turns out that none of the three functions F is primitive recursive, and each provides a layering of the class of primitive recursive functions. We have the following analogue of Proposition 4.8.

6.8 PROPOSITION. *Consider the three functions F of Theorem 6.7 where each is based on the successor function (that is f is successor). Each primitive recursive function g is eventually dominated by the 1-placed function*

$$F(i, 1, \cdot) \qquad F(i, \cdot) \qquad F(i, \cdot)$$

for some r (depending on g).

This result was first proved by Ackermann using the 3-placed function F. His proof used some rather convoluted approximations between values of functions (and is seems that this is the proof most often presented). Hilbert suggested that the function might be easier to handle if it was constructed in terms of a higher order operator, as in Theorem 6.7. The central function F is due independently to both R. M. Robinson and R. Péter. Again their proofs where rather convoluted. Part of their aim was to 'simplify' Ackermann's proof by combining the roles of the variables r and x. However, it is debatable if they did achieve their aims. The third function F is extracted from the work of Grzegorczyk, who was the first to describe the layering of $\mathcal{P}$. (The later part of section 4 is survey of Grzegorczyk's work.)

Proposition 6.8 can be modified along the lines of Proposition 4.10, and we find that essentially the same layering is obtained.

The Ackermann function F and its variants is not primitive recursive, but clearly not too complicated. In fact, this function F is often presented as an example of 'the next complexity beyond' primitive recursion, and it seems that some people believe this. Let's try to make this idea more precise. Consider the two clones

$$\mathcal{P}(0) = \mathcal{P} = \boldsymbol{P}(\emptyset) \qquad \mathcal{P}(1) = \boldsymbol{P}(F)$$

that is the clone of primitive recursive functions and the clone of those functions that are primitive recursive in F, the Ackermann function. The reason for the indexing will become clear shortly.

We have $\mathcal{P}(0) \subsetneq \mathcal{P}(1)$, and we can wonder what clones lie between these two.

Let $\lfloor\mathbb{Q}\rfloor$ be the set of rational number r with $0 \le r \le 1$. We extend the indexing used above.

6.9 PROPOSITION. *There is an indexed family of clones*

$$\big(\mathcal{P}(r) \mid r \in \lfloor\mathbb{Q}\rfloor\big)$$

where each is closed under primitive recursion and with

$$\mathcal{P}(0) \subseteq \mathcal{P}(r) \subsetneq \mathcal{P}(s) \subseteq \mathcal{P}(1)$$

for all $r, s \in \lfloor\mathbb{Q}\rfloor\big)$ *with* $r < s$.

In other words, there is a lot going on between primitive recursion and the complexity achieved by the Ackermann jump operator.

This higher level diagonalization method enables us to jump out of the primitive recursive grip of any bounding function. Given such a function f we may use and standard jump operator J to set

$$f^\star x = J^x f x \quad \text{or} \quad f^\star x = J^x f 1$$

or something similar to obtain a must faster bounding function $f^\star$, that is with $\mathcal{P}(f) \subsetneq \mathcal{P}(f^\star)$. This gives us a more powerful jump operator which we can use to jump to higher levels. Clearly, this idea can be repeated, but where will it take us? To give a hint of the answer to that we need to organize the method properly.

6.10 DEFINITION. The higher order operator

$$P : \mathbb{N}'' \longrightarrow \mathbb{N}''$$

is given by

$$PFfx = F^x f x$$

for each $F \in \mathbb{N}'', f \in \mathbb{N}'$ and $x \in \mathbb{N}$. ■

(The 'P' here stands for Péter who was, perhaps, the first person brave enough to venture into these cosmic regions.)

Using P we see that

$$f^\star = PJfx$$

(at least in the left hand case), and iterating $(\cdot)^\star$ is just a standard functional iteration of P. It is perhaps obvious what we are going to do next, but before that let's look at more complicated versions of the functions of Definition 6.5 obtained by introducing an extra variable into each one.

6.11 DEFINITION. Given a single data function $f \in \mathbb{N}'$ we specify several multi-placed functions.

$$\begin{array}{lll}
G(0,0,0,x) = fx & G(0,0,x) = fx & G(0,0,x) = fx \\
G(j',0,0,x) = G(j,x,0,x) & G(j',0,x) = G(j,x,x) & G(j',0,x) = 2 \\
G(j,i',0,x) = G(j,i,x,x) & G(j,i',0) = G(j,i,1) & G(j,i',0) = G(j,i,y) \\
G(j,i,r',x) = G(j,i,0,y) & G(j,i',x') = G(j,i,y) & G(j,i',x') = G(j,i,y) \\
\quad \text{where } y = G(j,i,r,x) & \quad \text{where } y = G(j,i',x) & \quad \text{where } y = G(j,i',x)
\end{array}$$

where $j,i,r,x \in \mathbb{N}$. Of course, there are three different functions G. ■

Handling the functions F of Definition 6.5 in terms of the first order recursive specifications is a bit tricky. Handling these functions is even trickier. Fortunately, there is an analogue of Theorem 6.7.

6.12 THEOREM. *For the functions specified in Examples 6.11 we have*

$$\begin{array}{lll}
G(j,i,r,x) = (A^i(H^j f))^{r+1}x & G(j,i,x) = R^i(T^j f)x & G(j,i,x) = B^i(C^j f)x \\
\quad \textit{where } H = PA & \quad \textit{where } T = PR & \quad \textit{where } C = PB
\end{array}$$

for the various cases.

The proof of this follows the same lines as that of Theorem 6.7, except the inductions are nested to a greater depth.

In these examples the iteration of P is always in conjunction with a jump operator. That is, iterations like $(PJ)^j$ are used. We can also use direct iterations of P.

6.13 EXAMPLE. Using the operators A, R, B, P and a function $f \in \mathbb{N}'$ set

$$H(k,j,r,x) = ((P^k A)^j f)^{r+1}x \quad H(k,j,x) = (P^k R)^j fx \quad H(k,j,x) = (P^k B)^j fx$$

for each $j,i,r,x \in \mathbb{N}$ to produce three different function H. ■

It turns out that each of these is considerably more powerful than the corresponding function G. You might like to work out a recursive specification of each.

The ω-chain of layers

$$\big(\mathcal{P}_r \,|\, r < \omega\big)$$

can be continued to produce much longer chains of clones capturing more an more complex functions, each of which can be generated using some forms of recursions.

Consider the general procedure set out earlier in this section.

We start for a suitable base function $f \in \mathbb{N}'$ and a jump operator $J \in \mathbb{N}''$. The ω-chain of functions

$$(\omega) \quad (J^i f \,|\, i < \omega)$$

control the layering of $\mathcal{P}$. Using the higher order operator $P \in \mathbb{N}'''$ we find that the function

$$x \longmapsto J^x f x = PJfx$$

jumps out of $\mathcal{P}$ and dominates each member of (ω). This requires just one use of P. We can iterate P to produce a second ω-chain

$$(\omega^\omega) \quad (P^i J f \,|\, i < \omega)$$

where the increase in complexity of each step is at the ω-strength (similar to the increase in complexity from f to PJf).

To go higher we lift the trick up a level. Consider the function $Q \in \mathbb{N}^{(iv)}$ given by

$$Q\phi F f x = \phi^x F f x$$

for each $\phi \in \mathbb{N}''', F \in \mathbb{N}'', f \in \mathbb{N}, x \in \mathbb{N}$. The function $QPJf$ out jumps each member of (ω^ω), and by iterating Q we produce a third ω-chain

$$(\omega^{\omega^\omega}) \quad (Q^i PJf \,|\, i < \omega)$$

where the increase in complexity of each step is at the ω^ω-strength.

It is now clear how this process should continue. By doing so we obtain all the functions that can be handle in first order peano arithmetic (the elementary functions in the other sense). However, to organize this properly we need to use ordinals, and that's another story.

Exercises

33 Consider Example 6.3. By viewing f in curried form show there is a rather simple function

$$\phi : \mathbb{N} \longrightarrow \text{Lists of } \mathbb{N}$$

which is independed of h, and which can be used to give an explicit description of f.

34 Complete the proof of Theorem 6.7, that is for the two other functions F.

35 Consider the 2-variable versions F of the Ackermann functions based on the successor function S. It is often stated that an explicit description of the function $F(i,\cdot)$ is difficult to give beyond the case $i=3$. This is certainly the case if the multi-recursive specifications are used.

Let $\daleth = \beth(\cdot, 2, 1)$.

(a) For the functions

$$f = RS,\ g = Rf = R^2S,\ h = Rg = R^3S,\ k = Rh = R^4S,\ l = Rk = R^5S$$

write down explicit descriptions of

$$S^r x \quad f^r x \quad 3 + g^r x \quad 3 + h^r x \quad 3 + k^r x$$

and hence show that

$$3 + R^5 x = \daleth^{3+x} 1$$

(for $x, r \in \mathbb{N}$).

(b) Show that

$$2 + B^5 x = \daleth^{2+x} 1$$

(for $x \in \mathbb{N}$).

(c) Can you perform a similar analysis using the jump operator A?

36 This exercise deals with the crucial part of the proof of Theorem 6.8. The whole proof is like that of Proposition 5.3 which is dealt with in Exercise 31. We need a method of passing across each of the relevant constructions. Here we deal with the step across an instance of primitive recursion (for all the other steps are straight forward.) Also we deal only with the step for the Ackermann jump A. The other jump operators are dealt with in the same way but can be slightly less convenenient.

Consider an instance of primitive recursion

$$\phi 0 p = \theta p \qquad \phi(x', p) = \psi(x, p, \phi x p)$$

which produces a function ϕ from two functions θ, ψ. Here the parameter p will be a tuple of natural numbers.

Suppose f is a bounding function with

$$\theta, \psi \leq_a f$$

for some start point a.

(a) Show that $\phi x \leq_a f^{x+1}$ for each $x \in \mathbb{N}$.

(b) Show that $\phi \leq_a Af$.

37 Set up the inductions which give a proof of Theorem 6.12.

38 Write down recursive specifications of the functions H in Example 6.13.

39 Find a function which is not primitive recursive but does have a recursive specification.

7 Feasibililty and computability

In this final section we look at how this material could be developed into a much longer account. We look at the two extremes of the topic, and how it relates to several other subjects.

We have seen that a recursive specification of a functions encodes an algorithm for evaluating that function. In fact, a specification is more correctly viewed as a description of an algorithm than of a function. How practical is such an algorithm? The answer depends on the particular specification.

A specification may be rather convoluted and still evaluate a rather simple function. The specification of Examples 2.6 is of this kind. The function produced is little more than linear and can be evaluated in a straight forward manner. Instances of the specifications of Definitions 6.5 and 6.11 and the of Examples 6.13 can provide even more dramatic illustrations of daft ways to evaluate simple functions (by allowing the base function f to be the identity function).

This is hinting at a topic we won't even touch upon here. How can we take a specification and transform it into a more efficient specification of the same function?

Some functions are inherently complicated, and any evaluation algorithm will be costly. We have seen some of these functions already, and later in this section we will see that there are even more costly functions. Before let's consider what a cheap function might be (or inexpensive as they say these days).

Each run of an algorithm uses some resources, either time or space or both.

Roughly speaking the time resource is the number of steps needed to complete the algorithm. For a single recursion this is something like the number of passes through the recursion. However, for nested recursion the measure is less straight forward. Clearly, if a run of an algorithm can take a long time to terminate then it is not very useful.

The space resource of an algorithm is not so obvious. As a run proceeds it may be necessary that certain pieces of data have to be stored for later use. Certain calculations may have to be supended to await the results of later calculations. A body recursion, as in Table 1 is a good illustration of this. To calculate $f(7,p)$, say, we must first use the data function k to calcuate and store

$$p^{(0)} = p,\ p^{(1)} = k(6,p^{(0)}),\ p^{(2)} = k(5,p^{(0)}),\ \dots,\ p^{(7)} = k(0,p^{(6)})$$

and then use these in reverse order to obtain

$$t_0 = f(0,p^{(7)}),\ t_1 = f(1,p^{(6)}),\ \dots,\ t_7 = f(7,p^{(0)})$$

with t_7 as the required value.

Even wilder things can happen, especially with the recursion contains some nesting of the target function.

7.1 EXAMPLE. Using the data functions

$$\mathbb{P} \xrightarrow{\ g\ } \mathbb{T} \qquad \mathbb{N},\mathbb{P},\mathbb{T} \xrightarrow{\ h\ } \mathbb{T} \qquad \mathbb{N},\mathbb{P} \xrightarrow{\ k\ } \mathbb{P} \qquad \mathbb{N},\mathbb{T} \xrightarrow{\ l\ } \mathbb{P}$$

consider the function

$$\mathbb{N},\mathbb{P} \xrightarrow{\ f\ } \mathbb{T}$$

specified by

$$f(0,p) = gp \qquad f(x',p) = h(x,p,t) \quad \text{where} \quad \begin{cases} t = f(x,r) \\ r = l(x,s) \\ s = f(x,q) \\ q = k(x,p) \end{cases}$$

for $x \in \mathbb{N}, p \in \mathbb{P}$. ■

You should consider how an evaluation of $f(3,p)$ could be organized.

Informally we say a function (or, strictly speaking, an algorithm) is feasible if it doesn't require too much space and time resources to calculate. This is usually taken to mean there are ploynomial bounds on the resources, but sometimes slightly more liberal bounds are used, and in some case even cubic bounds are beyond the pale. However, in all cases a feasible function belongs to the class $\mathcal{K}$ of Kalmar elementary functions. In fact, it belongs to the lower levels of this class.

As mentioned in section 5 the subject of computational complexity is concerned with the lower levels parts of $\mathcal{K}$, and the methods used are very different to the ones outlined here. By analogy we may say that computational complexity is to forms of recursions what numerical analysis is to calculus, and form of recursion is to recursion theory what calculus is to functional analysis.

In section 5 we introduced the notion of a bounded search, and showed (or rather, stated) that such a construction is weaker than $\boldsymbol{K}$. In particular, if h is in $\mathcal{K}$ or $\mathcal{P}$, then so is

$$(\mu y \leq x)[h(y,p) = 0]$$

(as a function of x and the parameters p).

The crucial aspect of bounded search is that a nominated input x of the constructed function that puts a bound of the length of the search. We know at the outset how many values of the data function we migh need. Of course, we can be more adventurous and try the same idea without this safety net.

7.2 DEFINITION. For a given data function h, as to the left,

$$\mathbb{N},\mathbb{P} \xrightarrow{\ h\ } \mathbb{N} \qquad\qquad \mathbb{P} \xrightarrow{\ f\ } \mathbb{N}$$

let f, as to the right, be the function given by

$$fp = \text{the least } y \text{ with } h(y,p) = 0 \text{ (if there is such a } y\text{)}$$

for $p \in \mathbb{P}$. We say f is obtained from h by unbounded search. ■

We extend the previous notation and write

$$(\mu y)[h(y,p) = 0]$$

for the constructed function f. This construction is often called μ-recursion (even though it is not a recursion in the true sense).

There is a crucial difference between this and every other construction we have looked at. Even if the data function h is total, the contructed function $(\mu y)[h(y,p) = 0]$ need not be total. Thus, to analyse the power of μ we must consider *partial functions*.

At this stage we should rework much of the prevous material in terms of partial functions. We don't need to do this here for we don't need the details, and in any case no fundamental changes are needed. Let's assume it has been done.

7.3 DEFINITION. The class of general recursive functions is the least clone of partial first order functions

$$\mathbb{N}^{1+s} \xrightarrow{\quad f \quad} \mathbb{N}$$

that contains sufficiently many initial functions and is closed under primitive recursion and unbounded search. ■

In this definition we didn't say what the initial functions should be. That is because the definition contains a red herring which such be smoked out.

7.4 THEOREM. *Provided we have access to a small number of simple functions, any use of primitive recursion can be rephrased in term of unbounded search.*

Sketch proof. We won't say what simple functions are needed, but they are little more than addition and multiplication. Using these and unbounded search we can produce a list coding function $\beta \in \mathbb{N}'$ with the following property.

> For each list
> $$[a(0), a(1), \dots, a(l)]$$
> from $\mathbb{N}$ there is some $a \in \mathbb{N}$ such that
> $$\beta(a,i) = a(i)$$
> for each $i \leq l$.

The construction of such a function β is not very difficult, but the details are not too enlightening. (The use of 'β' may seem a little strange. It was first used by Gödel and the custom seem to have stuck.)

Now consider an instance

$$f(0,p) = gp \qquad f(x',p) = h(x,p,f(x,p))$$

of primitive recursion. Let $a(\cdot,\cdot)$ be the function given by

$$a(x,p) = (\mu y)[\beta(y,0) = gp \text{ and } (\forall i < x)[\beta(y,i') = h(i,p,\beta(y,i))]]$$

for $x \in \mathbb{N}$ and parameters p. It is not too hard to show that the body of this unbounded search stays within the realm of acceptable functions. In particular, the function $a(\cdot,\cdot)$ can be built up from the initial functions by composition and unbounded search. But

$$\beta(a(x,p),i) = f(i,p)$$

for each $i \leq x$, and hence

$$f(x,p) = \beta(a(x,p),x)$$

to give the required result. ■

This result shows that the use of primitive recursion is not needed to build up the general recursive functions. In fact, a very similar proof shows that almost any form of recursion can be subsumed by unbounded search, and hence the class of general recursive functions must be quite large.

In 1936 Turing analysed the notion of 'computable' in an abstract setting. That is, he gave a precise description of what is computable, and so formulated a definition of that notion. (He did not analyse the related notion of 'computation'. He described what could be achieved by all possible computations, but he did not describe what all computations could look like. He showed that any computation could be broken down into small primitive steps, and then showed what any sequence of such steps could produce.)

When applied to the natural numbers this precise notion of computable gives exactly the general recursive functions. Furthermore, there have been many different analyses of what a 'computable' function over $\mathbb{N}$ could be, and all have produced the same class of functions.

Every function over $\mathbb{N}$ that we have seen in these notes is general recursive. Some of the functions we have used seem to be rather complicated. However, once we move into the realm of general recursive functions, we soon realize we haven't got anywhere near the limits of their complexity.

A rather crude way of measuring the complexity of a function uses the ordinals. We saw the beginnings of this idea in section 4. There we disected the class $\mathcal{P}$ of primitive recursive functions into an ω-chain of layers. In this sense we can say these functions have complexity $\leq \omega$. The methods of section 6 produced functions of complexity $\leq \omega^\omega$ and then of complexity $\leq \omega^{\omega^\omega}$.

Consider the sequence $\omega[\cdot]$ of odinals generated by

$$\omega[0] = 1 \qquad \omega[t'] = \omega^{\omega[r]}$$

for r, ω. Thus we have produce functions of complexities $\leq \omega[r]$ for $r = 1, 2, 3$. By iterating these methods we can produce functions of complexity $\leq \omega[r]$ for any $r < \omega$.

All of these construtions are carried out on a certain base function f.

Let

$$\epsilon_0 = \bigvee(\omega[r] \mid r <, \omega)$$

to produce the least ordinal θ larger than each $\omega[r]$. It is not too difficult to take a 'diagonlization' through the previous contructions and so produce a function $f^\star$ which is more complicated than any previous functions. This is a function of complexity ϵ_0.

But now we can repeat the whole of the previous constructions starting with $f^\star$ as a base function. By repeating this idea we obtain functions $f^\star, f^{\star\star}, f^{\star\star\star}$ of greater and greater complexity.

Consider the ordinals θ such that

$$\theta = \omega^\theta$$

hold. Thus ϵ_0 is the smallest such ordinal. For any ordinal ζ we may set

$$\zeta[0] = \zeta + 1 \qquad \zeta[r'] = \omega^{\zeta[r]}$$

for each $r < \omega$, and then

$$\zeta^\star = \bigvee(\zeta[r] \mid r <, \omega)$$

is the next θ strictly larger than ζ. In particular

$$\epsilon_0 = 0^\star$$

is the least such ordinal.

By iterating this construction we produce a long chain

$$\epsilon_0,\ \epsilon_1,\ \epsilon_2,\ \ldots,\ \epsilon_\omega,\ \ldots,\ \epsilon_{\epsilon_0},\ \ldots$$

of larger and larger ordinals θ. Formally we set

$$\epsilon_0 = 0^\star \qquad \epsilon_{\alpha+1} = \epsilon_\alpha^\star \qquad \epsilon_\mu = \bigvee\{\epsilon_\alpha \mid \alpha < \mu\}$$

for each ordinal α and limit ordinal μ.

By iterating the methods outlined above we can produce a function of complexity ϵ_α for any ordinal α you can name. Furthermore, each such function is general recursive.

It is an instructive exercise to see just how many ordinals you can name. If you can get beyond

$$\epsilon_{\epsilon_{\epsilon_{\ddots}}}$$

the least ordinal δ with

$$\delta = \epsilon_\delta$$

then you are doing well. But you still haven't got anywhere near the limits of the general recursive functions.

Some references

The following is a list of refrences related to this material. I did plan to attach a commentary on these but lack of time prevented me. However, even the raw refences may be useful.

References

[1] N.J. Cutland: *Computability, and introduction to recursive function theory,* C.U.P., 1980.

[2] M. Davis (ed): *The undecidable*, Raven Press, New York, 1965.

[3] H.B. Enderton: *A mathematical introduction to logic*, Academic Press, 1972.

[4] E.R. Griffor (ed): *Handbook of computability theory*, North Holland, 1999.

[5] L.A. Harington et al (ed): *Harvey Friedman's research on the foundations of mathematics*, North-Holland, 1985.

[6] E. Mendelson: *Introduction to mathematical logic*, Van Nostrand, 1964.

[7] P. Odifreddi: *Classical recursion theory*, North Holland, 1989.

[8] P. Odifreddi: *Classical recursion theory* vol II, North Holland, 1999.

[9] R. Péter: *Recursion Functions*, Academic Press, 1967.

[10] E.L. Post: Recursively enumerrable sets of positive intergers and their decision problems, *Bull. Am. Math. Soc.* 50 (1944) 184-316. Also reprinted in [2].

[11] H. Rogers, Jn: *Theory of recursive functions and effective computability*, McGraw-Hill, 1967.

[12] H.E. Rose: *Subrecursion: functions and hierarchies*, O.U.P., 1984.

[13] R.I. Soare: The history and concept of computability, pp 3–36 of [4].

[14] C. Smoryński: 'Big" news from Archimedes to Friedman, pp 353–366 of [5].

[15] C. Smoryński: Some rapidly growing functions, pp 367–380 of [5].

[16] C. Smoryński: The varieties of aborial experience, pp 381–397 of [5].

Some solutions

Section 1

1 Given $\phi(x)$ we assume (the quantified version of)

$$(\star) \quad (\forall y < x)\phi(y) \Longrightarrow \phi(x)$$

holds. Let $\psi(x)$ be $(\forall y < x)\phi(y)$ so that

$$(\forall x : \mathbb{N})\psi(x) \Longrightarrow (\forall x : \mathbb{N})\phi(x)$$

and

$$\psi(x+1) \Longleftrightarrow \psi(x) \wedge \phi(x)$$

hold (no matter what ϕ is). But $(\star)$ is just

$$\psi(x) \Longrightarrow \phi(x)$$

and hence

$$\psi(x+1) \Longleftrightarrow \psi(x)$$

which gives us a step case for ψ. Since $\psi(0)$ holds vacuously, we have $(\forall x : \mathbb{N})\psi(x)$ by a b/s induction, and this implies $(\forall x : \mathbb{N})\phi(x)$. ■

2 (b) $f(x, y, z) = h(y, z, \cdot)^x(g(y, z))$ ■

4 Consider the statement

$$[x] \quad (\forall i : \mathbb{N}) \left[f(x, i) = \left\{ \begin{array}{cl} 0 & \text{if } x < i \\ \binom{x}{i} & \text{if } i \leq x \end{array} \right\} \right]$$

for $x \in \mathbb{N}$. We verify $(\forall x : \mathbb{N})[x]$ by a b/s induction over x. The base case, $x = 0$, is immediate. The step case, $x \mapsto x'$, looks as though an inner induction over i is needed. However, it is sufficient to consider the cases $i = 0$ and $i \neq 0$ separately. ■

Section 2

6 (c) Fix y, z and let

$$\langle x \rangle \quad \text{abbreviate} \quad [H(x', y, z) = yH(x, y, z)]$$

so we require $(\forall x)\langle x \rangle$. This is proved by induction over x. Thus

$$\begin{array}{rcl} H(x'', y, z) & = & H(x', y, z)(y + x') \\ & = & yH(x, y', z)(y + x') \\ & = & yH(x, y', z)(y' + x) \quad = \quad yH(x', y', z) \end{array}$$

gives the induction step $x \mapsto x'$. The second equality uses the induction hypothesis, and the first and third uses the specification of H.

Next for a fixed z let

$$\langle\langle x\rangle\rangle \quad \text{abbreviate} \quad (\forall y)[H(x,y,z) = B(x,y,z)]$$

so we require $(\forall x)\langle\langle x\rangle\rangle$. This is proved by induction over x using the first part. Thus, for arbitrary y,

$$H(x',y,z) = yH(x,y',z) = yB(x,y',z) = B(x',y,z)$$

gives the induction step $x \mapsto x'$. The first equality uses the first part, the second uses the induction hypothesis, and the third uses the spec of B.

(d) Fix z and let

$$\langle x\rangle \quad \text{abbreviate} \quad (\forall y)[B(x',y,z) = B(x,y,z)(y+x)]$$

so we require $(\forall x)\langle x\rangle$. This is proved by induction over x. Thus

$$\begin{aligned} B(x'',y,z) &= yB(x',y',z) \\ &= yB(x,y',z)(y'+x) \\ &= yB(x,y',z)(y+x') \quad = \quad B(x',y,z)(y+x') \end{aligned}$$

gives the induction step $x \mapsto x'$. The second equality uses the induction hypothesis, and the first and last uses the specification of B.

Next for a fixed y, z let

$$\langle\langle x\rangle\rangle \quad \text{abbreviate} \quad [H(x,y,z) = B(x,y,z)]$$

so we require $(\forall x)\langle\langle x\rangle\rangle$. This is proved by induction over x using the first part. Thus

$$B(x',y,z) = B(x,y,z)(y+x) = H(x,y,z)(y+x) = H(x',y,z)$$

gives the induction step $x \mapsto x'$. The first equality uses the first part, the second uses the induction hypothesis, and the third uses the specification of H.

(e) Throughout all the steps the parameter z can be fixed. However, there are some parts where y can be fixed and some part where y must be allowed to vary (by the use of universla quantification). In general, a head recursion allows the parameters to be fixed but a tail recursion requires a variation of the parameters. I believe that this is why some people think that primitive recursion is primitive; the corresponding inductions are easier.

(f) Fix y, z and let

$$\langle x\rangle \quad \text{abbreviate} \quad (\forall u)[T(x, y+u, H(u,y,z)) = H(x+u,y,z)]$$

so we require $(\forall x)\langle x\rangle$ which is proved by induction over x. With $w = H(u,y,z)$

$$\begin{aligned} T(x',y+u,w) &= T(x,(y+u)',w(y+u)) \\ &= T(x,y+u',H(u',y,z)) \\ &= H(x+u',y,z) \qquad = \quad H(x'+u,y,z) \end{aligned}$$

gives the induction step $x \mapsto x'$. The third equality uses the induction hypothesis, and the others use the specs of H and T.

Now, since $H(0, y, z) = z$, setting $u = 0$ in this identity gives

$$T(x, y, z) = H(x, y, z)$$

as required.

(g) All three functions F are

$$F(x, y, z) = \frac{(x + y)!}{y!} \times \frac{y}{x + y} \times z$$

that is

$$F(0, y, z) = z \qquad F(x', y, z) = zy(y + 1) \cdots (y + x)$$

in a more understanable form. ■

7 The two specification look a bit better if we write ϕ is curried form. We then see that

$$h(x, s, t) = (\phi s)^x t = T(x, s, t)$$

follows by induction over x.

Even without this insight we can obtain the required equality. We first prove

$$H(x, s, \phi(s, t)) = \phi(s, H(x, s, t)) \qquad T(x, s, \phi(s, t)) = \phi(s, T(x, s, t))$$

both by induction over x. With these a second induction over x gives the required result. ■

10 We have

$$\mathrm{boris}(x, y, z) = xy + z$$

for $x, y, z \in \mathbb{N}$. This is proved by a progressive induction over x with variation of the parameters y, z. The progression is obtained by setting

$$x = 2u + i$$

where $i = 0, 1$. (This algorithm is known as Russian multiplication if P.C. Cobblers, the community copper, will allow us to say that these days.) ■

11 This generates a variant of Pascal's triangle but written out on the lattice points of the positive quarter of the {x,y,}-plane (which is perhaps more sensible. As a descent measure the sum $x + y$ will work.

To prove the specification has a unique solution, suppose f is any solution and show

$$(\forall x, y) \left[x + y \le z \Longrightarrow f(x, y) = \binom{x + y}{x} \right]$$

holds for all z. A b/s induction works, but the step argument splits into cases. ■

12 (a) It is convenient to consider four ranges of values of x.

Range of x	d	x^+	i^+	d^+
$121 < x$	$21i$	$x - 10$	$i - 1$	$d - 21$
$111 < x \le 121$	$21i$	$x - 10$	$i - 1$	(??)
$100 < x \le 111$	$21i + 2(111 - x)$	$x - 10$	$i - 1$	$d - 1$
$x \le 100$	$21i + 2(111 - x)$	$x + 11$	$i + 1$	$d - 1$

This table list most of the relevent values. Only the value

$$d^+ = 21i^+ + 2(111 - x^+)$$

for one segment is missing. But in this case we have

$$x^+ = x - 10 \quad i^+ = i - 1$$

so that

$$d^+ = d - 21 + 2e$$

where $e = 121 - x$ is the excess. We have $0 \leq e < 10$ from which the required $d - 21 \leq d^+ < d$ follows.

This observation enables us to argue by progressive induction using d as a descent measure.

Suppose the algorithm was left to run for ever by repeatedly passing through the recursion step. At each step d decreases by at leas 1, and so eventually will become negative and continue to decrease. Since x is always positive this means that eventually i will become negative. But at each step i changes by 1, and so to get from positive to negative must passs through 0. At that point the guard in the recursion ensures that the algorithm does stop.

A more succinct version of this is as follows. Suppose for some inputs the embedded algorithm does not terminate. Then there is some non-terminating input (x, i) where the measure d is as small as possible. One pass through the recursion decreases the measure to $d^+ < d$ and, choice of d, from that position the algorithm does terminate, which is a contradiction.

A rephrasing of this argument shows that the specification has a unique solution, and this solution is total.

(b) We verify the description for $i \neq 0$, again using a progressive induction over d. However, there are sevearl cases that must be considered.

Suppose first that $90 \leq x \leq 100$ with $i \neq 0$. Then

$$f(x, i) = f(x + 11, i + 1) = f(x + 1, i)$$

since $101 \leq x$. Repeating this sufficiently many times give

$$f(x, i) = f(100, i) = f(111, i + 1) = f(101, i) = f(91, i - 1)$$

and (since 91 is one possible x) we may use a descent on the index i to get

$$f(x, i) = f(91, i - 1) = f(91, 0) = 91$$

as required for these inputs (x, i). Notice that we have shown

$$f(101, i) = 91$$

for $i \neq 0$.

Now consider any $x \leq 100$ with $i \neq 0$. WE have

$$f(x, i) = f(x + 11, i + 1) = f(x + 1, i)$$

and then by iteration

$$f(x,i) = f(x + 11(r+1), i + r + 1)$$

for r where

$$x + 11r \leq 100 < x + 11r + 11$$

holds. But now

$$90 \leq x + 11r + 1 \leq 101$$

and hence the previous calculations give

$$f(x,i) = f(x + 11(r+1), i + r + 1) = 91$$

as required.

Finally suppose $100 < x$ with $i \neq 0$. Then

$$f(x,i) = f(x - 10, i - 1)$$

and

$$90 + 10(i-1) \leq x - 10 \iff 90 + 10i \leq x$$

so that a use of the induction hypothesis gives

$$\begin{aligned} f(x,i) &= f(x-10, i-1) \\ &= \begin{cases} (x-10) - 10(i-1) & \text{if } 90 + 10(i-1) \leq (x-10) \\ 91 & \text{if } (x-10) < 90 + 10(i-1) \end{cases} \\ &= \begin{cases} x - 10i & \text{if } 90 + 10i \leq x \\ 91 & \text{if } x < 90 + 10i \end{cases} \end{aligned}$$

as required.

There is nothing special about this exercise, except that it shows that some specifications can be unecessarily complicated. ■

13 (d) Nobody knows whether or not this function is total. ■

Section 3

16 For both algorithms we have

$$a(0,y) = 1 \qquad a(x,y) = 1 + a(x,y)$$

for which

$$a(x,y) = 1 + x$$

is the unique solution. ■

17 Consider a run from $M(\mathsf{S}\ulcorner x \urcorner, \ulcorner y \urcorner)$ for the two algithms. Indicating the number of steps above each transition sequence we have

$$M_l(\mathsf{S}\ulcorner x \urcorner, \ulcorner y \urcorner) \Longrightarrow A(M_l(\ulcorner x \urcorner, \ulcorner y \urcorner), \ulcorner y \urcorner) \xrightarrow{m(x,y)} A(\ulcorner y \times x \urcorner, \ulcorner y \urcorner) \xrightarrow{1+xy} \ulcorner y + y \times x \urcorner$$

$$M_r(\mathsf{S}\ulcorner x \urcorner, \ulcorner y \urcorner) \Longrightarrow A(\ulcorner y \urcorner, M_l(\ulcorner x \urcorner, \ulcorner y \urcorner)) \xrightarrow{m(x,y)} A(\ulcorner y \urcorner, \ulcorner y \times x \urcorner) \xrightarrow{1+y} \ulcorner y + y \times x \urcorner$$

for the two cases. Thus

$$m_l(0,y) = 1 \qquad m_r(0,y) = 1$$
$$m_l(x',y) = m(x,y) + xy + 2 \qquad m_r(x',y) = m(x,y) + y + 2$$

are the two specifications. These produce

$$m_l(x,y) = \frac{x(x-1)y}{2} + 2x + 1 \qquad m_r(x,y) = xy + 2x + 1$$

repectively. The slowness of M_l is probably caused by the substitution into the recursion variable when the algorithm A is invoked.

(As an operation addition is commutative, as an algorithm it is not.) ■

18 It is often useful to introduce an extra variable to store some transient value. In this case we use

$$M_L(0,y,z) \quad = z \qquad M_R(0,y,z) \quad = z$$
$$M_R(\mathsf{S}x,y,z) = M_L(x,y,A(z,y)) \qquad M_L(\mathsf{S}x,y,z) = M_L(x,y,A(y,z))$$

for $x,y,z \in \ulcorner\mathbb{N}\urcorner$. We find that

$$M_L(\ulcorner x\urcorner,\ulcorner y\urcorner,\ulcorner z\urcorner) = \ulcorner xy+z\urcorner = M_R(\ulcorner x\urcorner,\ulcorner y\urcorner,\ulcorner z\urcorner)$$

this time for $x,y,z \in \mathbb{N}$.

For these algorithms the costs are give by

$$m_L(0,y,z) = 1 \qquad m_R(0,y,z) = 1$$
$$m_L(x',y) \quad = m(x,y,y+z) + z + 2 \qquad m_R(x',y) \quad = m(x,y,y+z) + y + 2$$

respectively. These look vary similar, but the solutions are

$$m_L(x,y,z) = \frac{x(x-1)y}{2} + x(z+2) + 1 \qquad m_R(x,y,z) = xy + x(z+2) + 1$$

and as with M_l, M_r we have a quadractic descrepancy. ■

Section 4

20 We have

$$(a) \quad fx = 2 + x \qquad (m) \quad gx = 2 \times x \qquad (e) \quad hx = 2^x$$

for each x, and hence

$$(a) \quad f^r x = 2r + x \qquad (m) \quad g^r x = 2^r \times x \qquad (e) \quad h^r x = \beth(r,2,x)$$

for each r. This gives

$$(a) \quad f^+ x = 2x + 1 \qquad (m) \quad g^+ x = 2^x \qquad (e) \quad h^+ x = \beth(x,2,1)$$

and hence $g^+ = h$.

This shows that the function hierarchies based on g, h are essentially the same but the indexes are out of step by 1.

To deal with f we have

$$1 + f^{+}x = 2(1 + x)$$

so that

$$1 + (f^{+})^{r}x = 2^{r}(1 + x)$$

and hence

$$1 + f^{++}x = 2^{x} \times 2 = 2^{1+x} = h(1 + x)$$

for each x. Thus the function hierarchies based on g, h are essentially the same but the indexes are out of step by 2 and there is an internal an external shift by 1.

These kinds of discrepancies always occur when comparing functions hierarchies. Of course, depending on how fine we wish to measure complexity, it has to be decided what kind of discrepancy is acceptable. ■

21 We have

$$f^{+}0 = 1 \qquad f^{+}x' = f(f^{+}x) = f^{+}(fx)$$

which is an instance of primitive recursion and an instance of tail recursion.

Assumming f is a bounding function we show

$$y + f^{+}x \leq f^{+}(x + y)$$

by induction on y. The base case is trivial, and the step follows since

$$f^{+}(x + y + 1) = f(f^{+}(x + y)) \geq f(y + f^{+}x) \geq 1 + y + f^{+}x$$

using the given properties of f. The comparison shows immediately that f is monotone, and by setting $x = 0$ it shows that f is strictly inflationary.

An easy calculation shows that $S^{+} = S$ and hence $S^{[r]}x = Sx$. Thus the 2-placed function $f^{[r]}x$ can be primitive recursive in f. (The fact that $S^{+} = S$ is one of the reasons why $(\cdot)^{+}$ is not a good jump operator.) However, if f does manage to eventually get away from S, then each function which is primitive rcursive in f is eventually dominated by some $f^{[r]}$. In particular, if the 2-placed function is primitive recursive in f then there is some s such that

$$f^{[x]}x \leq f^{[s]}x$$

holds for all sufficiently large x. Taking x somewhat larger than s leads to a contradiction. (This is typical of the way a jump operator is used to escape from primitive recursion. The details in this case are a bit fiddly because of the nature of $(\cdot)^{+}$.) ■

25 (a,b) Consider the lattice points in the positive quadrant of the (u, v)-plane.

Diagram in here

These points can be enumerated by moving down each skew diagonal

$$u + v = d \quad (d = 0, 1, 2, \ldots)$$

in turn.

On the line with index d there are $1 + d$ points. The point (x, y) lies on the line with index $d = x + y$, and is the $(x + 1)^{\text{st}}$ point from the top. Thus, remembering that $P(0, 0) = 0$ (that is, we start enumerating from 0 and not 1) we have

$$1 + p(x, y) = \left(\sum_{d < x+y} (i + d) \right) + (x + 1)$$

so that

$$2P(x, y) = (x + y + 1)(x + y) + 2x$$

to give the required result.

(c,d) Given $z \in \mathbb{N}$ we want to solve

$$2z = (x + y + 1)(x + y) + 2x$$

for x, y. We first find $d = x + y$ and so determine the skew diagonal on which the required point lies. We know that

$$(d + 1)d \leq 2z \leq (d + 2)(d + 1)$$

and hence

$$\delta z = (\mu w < z)[2z < (w + 2)(w + 1)]$$

give the index of the line. Observe that $\delta \in \mathcal{K}$.

Let

$$\Delta z = ((1 + \delta z)\delta z)/2$$

so that Δ counts the number of points on the previous shew diagonals, that is

$$\delta z = x + y \qquad z = \Delta z + x$$

give x, y. Thus

$$Lz = z - \Delta z \qquad Rz = \delta z - Lx$$

give the required functions. ■

Section 5

28 We find that

$$\text{Neg}x = D(1, x) \qquad \text{Eq}(x, y) = \text{Neg}^2(D(x, y) + D(ymx))$$

will do. ■

30 (a) First of all observe that each component value

$$\text{Neg}^2(z) \quad \text{Neg}(\text{Eq}(z, 1)) \quad \text{Neg}(\text{Eq}(z, yx))$$

is either 0 or 1, and hence this P is a $\{0,1\}$-functions. But now

$$P(z) = 1 \iff \begin{Bmatrix} \text{Neg}^2(z) = 1 \\ \text{Neg}(\text{Eq}(z,1)) = 1 \\ \prod_{y<z}\prod_{x<z} \text{Neg}(\text{Eq}(z,yx)) = 1 \end{Bmatrix}$$

$$\iff \begin{Bmatrix} z \neq 0 \\ z \neq 1 \\ (\forall y < z, x < z)[z \neq yx] \end{Bmatrix} \qquad \iff z \text{ is prime}$$

to give the required result.

(b) The function

$$\pi(x) = \sum_{i<x} P(i)$$

has the required property.

(c) Consider the decision function h given by

$$h(x,z) = \text{Neg}(P(z) \times \text{Neg}(\text{Eq}(\pi(z),x)))$$

so that

$$\begin{aligned} h(x,z) = 0 &\iff P(z) \times \text{Neg}(\text{Eq}(\pi(z),x)) = 1 \\ &\iff P(z) = 1 \text{ and } \text{Eq}(\pi(z),x) = 0 \\ &\iff z \text{ is prime and} \pi(z) = x \qquad \iff z = p_x \end{aligned}$$

and we may test for the required value. Now consider the function f given by

$$f(x,y) = \mu(z < y)[h(x,z) = 0]$$

with a bounding input y. Certainly we have $f \in \mathcal{K}$, and from above we see that $f(x,y) = p_x$ *provided* $p_x < y$. Thus it suffices to use an elementary bound on p_x. Since 2^{2^x} will do, we have

$$p_x = f(x, 2^{2^x})$$

to give the required result.

(You can see from this kind of argument that in a full development of the properties of $\mathcal{K}$ one of the first things to do is produce some numerical gadgets for handling boolean combinations and bounded quantifiers.) ■

31 Let f be a bounding function with the extra properties.

(b) Suppose the function k is dominated by f. Then with $y = \max\{x,p\}$ we have

$$\sum_{i<x} k(i,p) \leq x \times fy \leq f(y)^2 \leq f^2 y$$

using the properties of f.

(b) Suppose the function k is dominated by f. Then with $y = \max\{x,p\}$ we have

$$\prod_{i<x} k(i,p)(fy)^x \leq (fy)^{fy} \leq f(fy) = f^2 y$$

using the properties of f.

(d) The function f given by

$$fx = \beth(2,2,x) = 2^{2^x}$$

as the required properties, and hence so does each of its non-zero iterates. Thus, by an induction over its constyruction, for each $g \in \mathcal{K}$ there is some $r \in \mathbb{N}$ such that g is dominated by

$$f^r = \beth(2r, 2, \cdot)$$

as required. ∎

32 We have

$$\mathrm{Neg}^2(h(i,p)) = \begin{cases} 1 \text{ if } h(i,p) \neq 0 \\ 0 \text{ if } h(i,p) = 0 \end{cases}$$

and hence k(x,p) is a product of 0s and 1s. In fact $k(\cdot,p)$ is a step function starting of with value 1 and then dropping to value 0 (or remaining at 1). Let s be the least solution of $h(s,p) = 0$ (if there is such a solution). Then

$$k(x,p) = \begin{cases} 0 \text{ if } s < x \\ 1 \text{ if } x \leq s \end{cases}$$

by a simple argument. If there is no such s then $k(\cdot,p)$ is the constant 1 function. Next we observe that

$$f(x,p) = k(1,p) + \cdots + k(x',p)$$

where there are x components, all 1 or 0, and if any component is 0 then all the following ones are. By considering the relative positions of x and s we have

$$f(x,p) = \begin{cases} s \text{ if } s \leq x \\ 1 \text{ if } x < s \end{cases}$$

which is what we want. ∎

> Much of the initial development of the Kalmár consists of several tricks such as the ones used in these soloutions. This tends to obscure what is going on.

Section 6

33 It is easier to work in curried form. Thus

$$f0 = id \qquad fx' = hx \circ fx \circ fx$$

is a rephrasing of the specification. This immediately gives a better idea of what is going on.

Consider the function

$$\phi : \mathbb{N} \longrightarrow \text{Lists of } \mathbb{N}$$

(to lists of natural numbers) given by

$$\phi 0 = [\,] \qquad \phi x' = x^\frown \phi x^\frown \phi x$$

(for $x \in \mathbb{N}$). Here $[\,]$ is the empty list. At the recursion step, $x \mapsto x'$, the previous list is concatenated with itself and then x is appended to the front. Thus

$$\begin{aligned}
&f0 = [\,] \\
&f1 = [0] \\
&f2 = [1, 0, 0] \\
&f3 = [2, 1, 0, 0, 1, 0, 0] \\
&\vdots
\end{aligned}$$

and so on. Notice that

$$1 + \text{length}(\phi x) = 2^x$$

give the length of the generated list.

Given an arbitrary list

$$\ell = [a_1, \ldots, a_l]$$

of length l let

$$h\ell = ha_1 \circ \cdots \circ ha_l$$

formed by composing along the list. In particular $h[\,] = id$ and $h[a] = ha$.

Using this notation it is now easy to check that

$$fx = h(\phi x)$$

holds.

This solution illustrates two points.

When trying to convert a convoluted recursion into somekind of explicit description (or write a pseudo-program to evaluate the function) the inital aim should *not* be to produce valuesd of the function. Almost always the trick is to describe a 'template of evaluation', a description of how an evaluation will pan out and what extra information has to be found and stored. Once this has been generated the required value is easy to find.

This is precisely what Pascal spotted which enabled him to generate his triangle.

Following on from the first point, any decent study of recursion should *not* be restricted to functions over $\mathbb{N}$. Evaluations need other kinds of data structure, such as list and trees and so on. These should be allows as inputs and ouputs of functions. More importantly, recursions over these structure should be allowed.

Almost universally any presentation of this kind of material codes the elements of these other data structures as natural numbers. Try coding the list ℓ above as

$$|\ell| = p_1^{a_1} \times \cdots \times p_l$$

say, and then see what a mess the function ϕ becomes. ■

34 Consider the central function F. We proceed by a double induction over the recursion variables i, x. To help with this let

$$\begin{aligned}
&[i, x] \text{ abbreviate } F(i, x) = R^i f x \\
&[i, \cdot] \text{ abbreviate } (\forall x : \mathbb{N})[i, x] \\
&[\cdot, \cdot] \text{ abbreviate } (\forall i, x : \mathbb{N})[i, x]
\end{aligned}$$

so that $[\cdot,\cdot]$ is the eventually objective. The three clauses of the specification give

$$(1)\quad [0,0] \qquad (2)\quad [i,1] \Longrightarrow [i',0] \qquad (3)\quad [i,\cdot] \wedge [i',x] \Longrightarrow [i',x']$$

respectively. Part (1) is trivial, and since

$$Rg0 = g1$$

part (2) is immediate. Part (3) has a slight trick in it. Let

$$g = R^i f \qquad y = F(i',x)$$

so that

$$y = R^{i+1}fx = Rgx = g^{x+1}1$$

by the hypothesis $[i',x]$. But now, using the other hypothesis in the form $[i,y]$ we have

$$F(i',x') = F(i,y) = R^i fy = gy = g^{x+2}1 = Rgx' = R^{i+1}fx'$$

as required.

From (3) and induction over x gives

$$[i,\cdot] \wedge [i',0] \Longrightarrow [i',\cdot]$$

so that

$$[i,\cdot] \Longrightarrow [i',\cdot]$$

follows by (2). An induction over i now gives

$$[0,\cdot] \Longrightarrow [\cdot,\cdot]$$

so the required result follows by (1).

The right-hand F is dealt with in the same way. The only minor difference is that

$$Bg0 = 2$$

for each $g \in \mathbb{N}$. ■

35 (a) For each $x, r \in \mathbb{N}$ we have

$$\begin{aligned}
S^r x &= x + r \\
fx &= S^{x+1}1 && = x + 2 \\
f^r x &= x + 2r \\
gx &= f^{x+1}1 && = 2x + 3 \\
3 + gx &= 2(3 + x) \\
3 + g^r x &= 2^r(3 + x) \\
3 + hx &= g^{x+1}1 && = 2^{x+1} \times 4 \qquad = 2^{3+x} \\
3 + h^r x &= \beth(r, 2, 3 + x) \\
3 + kx &= h^{x+1}1 && = \beth(x + 1, 2, 4) = \beth(3 + x, 2, 1) = \daleth(3 + x) \\
3 + k^r x &= \daleth^r(3 + x)
\end{aligned}$$

and hence
$$3 + R^5S = 3 + k^{x+1}1 = \daleth^{x+1}4 = \daleth^{3+x}1$$
since $\daleth^2 1 = 4$.

(b) This calculation follows exactly the same pattern.

(c) Using similar calculations we have
$$ASx = 2x + 1 \qquad 1 + A^2Sx = 2^{1+x}x$$
for each x. The multiplier x makes it difficult to continue with precise identities. However, we do have
$$AS \sim R^2S \sim B^2S \quad A^2S \sim R^3S \sim B^3S \quad A^3S \sim R^4S \sim B^4S \sim \daleth$$
where '$\sim$'means the functions have approximately the same rate of growth. ∎

36 (a) The given bounding conditions $\theta, \psi \leq_a f$ ensure that
$$\left.\begin{matrix} p \leq y \\ a \leq y \end{matrix}\right\} \Longrightarrow \theta p \leq fy \qquad \left.\begin{matrix} x \leq z \\ p \leq z \\ t \leq z \\ a \leq z \end{matrix}\right\} \Longrightarrow \psi(x, p, t) \leq fy$$
hold for all $x, y, z, t \in \mathbb{N}$ and parameters p (where the comparision $p \leq y$ is read in the obvious manner). We fix p and show
$$\langle x \rangle \quad (\forall y) \left[\left\{\begin{matrix} x \leq y \\ p \leq y \\ a \leq y \end{matrix}\right\} \Longrightarrow \phi x p \leq f^{x+1}y \right]$$
by induction on x.

For the base case, $x = 0$, we have
$$\left.\begin{matrix} p \leq y \\ a \leq y \end{matrix}\right\} \Longrightarrow \phi 0 p = \theta p \leq fy$$
using the left hand condition on θ.

For the step case, $x \mapsto x'$, suppose
$$x \leq y \quad p \leq y \quad a \leq y$$
and let
$$t = \phi x p \qquad z = f^{x+1}y$$
so that
$$t \leq z \qquad y \leq z$$
using the induction hypothesis and the properties of f. The right hand condition on ψ gives
$$\phi x' p = \psi(x, p, t) \leq fz = f^{x+1}y$$
as required.

(b) Assuming

$$x \le y \quad p \le y \quad a \le y$$

part (a) gives

$$\phi x p \le f^{x+1} y \le f^{y+1} y = Afy$$

and hence $\phi \le_a Af$, as required. ■

37 To deal with the left hand function G we proceed by a triple induction over the recursion variables j, i, r. Let

$$\begin{array}{ll}
[j,i,r] & \text{abbreviate } (\forall x : \mathbb{N})[G(j,i,r,x) = (A^i(H^j f))^{r+1} x] \\
[j,\cdot,r] & \text{abbreviate } (\forall i : \mathbb{N})[j,i,r] \\
[j,i,\cdot] & \text{abbreviate } (\forall r : \mathbb{N})[j,i,r] \\
[\cdot,i,r] & \text{abbreviate } (\forall j : \mathbb{N})[j,i,r] \\
[\cdot,\cdot,r] & \text{abbreviate } (\forall j,i : \mathbb{N})[j,i,r] \\
[\cdot,\cdot,\cdot] & \text{abbreviate } (\forall j,i,r : \mathbb{N})[j,i,r]
\end{array}$$

so that $[\cdot,\cdot,\cdot]$ is the eventually objective. The four clauses of the specification give

$$\begin{array}{lr}
(1) & [0,0,0] \\
(2) & [j,\cdot,0] \Longrightarrow [j',0,0] \\
(3) & [j,i,\cdot] \Longrightarrow [j',i,0] \\
(4) & [j,i,0] \wedge [j,i,r] \Longrightarrow [j,i,r']
\end{array}$$

respectively.

From (4) and induction over r gives

$$(4^+) \quad [j,i,0] \Longrightarrow [j,i,\cdot]$$

so that

$$[j,i,0] \Longrightarrow [j,i',0]$$

follows by (3). Using this an induction over i gives

$$(3^+) \quad [j,0,0] \Longrightarrow [j,\cdot,0]$$

so that

$$[j,0,0] \Longrightarrow [j',0,0]$$

follows by (2). A final induction over j gives

$$[\cdot,0,0]$$

so that

$$[\cdot,\cdot,0]$$

follows by (3^+), and at last the required

$$[\cdot,\cdot,\cdot]$$

follows by (4^+).

To deal with the central function G we proceed by a triple induction over the recursion variables j, i, x. Let

$$\begin{array}{ll} [j,i,x] & \text{abbreviate } G(j,i,x) = R^i(T^j f)x] \\ [j,\cdot,r] & \text{abbreviate } (\forall i : \mathbb{N})[j,i,r] \\ [j,i,\cdot] & \text{abbreviate } (\forall r : \mathbb{N})[j,i,r] \\ & \vdots \end{array}$$

and so on, so that $[\cdot,\cdot,\cdot]$ is the eventually objective. The four clauses of the specification give

$$\begin{array}{lr} (1) & [0,0,\cdot] \\ (2) & [j,\cdot,\cdot] \Longrightarrow [j',0,\cdot] \\ (3) & [j,i,1] \Longrightarrow [j,i',0] \\ (4) & [j,i,\cdot] \wedge [j,i',x] \Longrightarrow [j,i',x'] \end{array}$$

respectively. The verification of these is little different to the verification of the clauses for the corresponding 2-variable function F.

From (4) and induction over x guves

$$[j,i,\cdot] \wedge [j,i',0] \Longrightarrow [j,i',\cdot]$$

so that

$$[j,i,\cdot] \Longrightarrow [j,i',\cdot]$$

follows by (3). Using this an induction over i gives

$$(\star) \quad [j,0,\cdot] \Longrightarrow [j,\cdot,\cdot]$$

so that

$$[j,0,\cdot] \Longrightarrow [j',0,\cdot]$$

follows by (2). Using this and (1) an induction over j gives

$$[\cdot,0,\cdot]$$

and now $(\star)$ gives $[\cdot,\cdot,\cdot]$, as required.

The proof for the right hand-function G is almost the same. ■

38 The trick here is that we can not fix the base function at the outset. We must treat it as an input to the function H.

To deal with the left-hand case consider the function

$$\mathbb{N}^4, \mathbb{N}' \xrightarrow{\quad H \quad} \mathbb{N}$$

specifiedation by

$$\begin{aligned} H(k,0,0,x,f) &= fx \\ H(0,j',0,x,f) &= H(0,j,x,x,f) \\ H(k',j',0,x,f) &= H(k,x,0,x,g) \\ \text{where } g &= H(k',j,0,\cdot,f) \\ H(k,j,r',x,f) &= H(k,j,0,y,f) \\ \text{where } y &= H(k,j,r,x,f) \end{aligned}$$

for $k, j, r, x \in \mathbb{N}, f \in \mathbb{N}'$. Let

$$\begin{array}{ll} [k,j,r] & \text{abbreviate } (\forall x : \mathbb{N}, f : \mathbb{N}')[H(k,j,r,x,f) = ((P^k A)^j f)^{r+1} x] \\ [k,j,\cdot] & \text{abbreviate } (\forall r : \mathbb{N})[k,j,r] \\ & \vdots \end{array}$$

and so on. The four clauses of the specification give

$$\begin{array}{lr} (1) & [\cdot,0,0] \\ (2) & [0,j,\cdot] \Longrightarrow [0,j',\cdot] \\ (3) & [k,\cdot,0] \wedge [k',j,0] \Longrightarrow [k',j',0] \\ (4) & [k,j,0] \wedge [k,j,r] \Longrightarrow [k,j,r'] \end{array}$$

respectively. These are verified in the usual way.

We use (1,2,3,4) to show $[\cdot,\cdot,\cdot]$ by a quadruple induction.

From (2) an induction over j gives

$$(2^+) \quad [0,0,\cdot] \Longrightarrow [0,\cdot,\cdot]$$

which we will use later.

From (3) a second induction over j gives

$$[k,\cdot,0] \wedge [k',0,0] \Longrightarrow [k',\cdot,0]$$

and hence

$$[k,\cdot,0] \Longrightarrow [k',\cdot,0]$$

follows by (1). An induction over k gives

$$[0,\cdot,0] \Longrightarrow [\cdot,\cdot,0]$$

and hence

$$(\star) \quad [0,0,\cdot] \Longrightarrow [\cdot,\cdot,0]$$

follows by (2^+).

From (4) and induction over r gives

$$(4^+) \quad [k,j,0] \Longrightarrow [k,j,\cdot]$$

and hence

$$[0,0,\cdot] \Longrightarrow [\cdot,\cdot,\cdot]$$

follows by combining $(\star)$ and (4^+).

Finally, from (1) and (4^+) we have $[\cdot,0,0]$ and so the last previous implication gives the required result.

To deal with the central case consider the function

$$\mathbb{N}^3, \mathbb{N}' \xrightarrow{\quad H \quad} \mathbb{N}$$

specified by

$$
\begin{aligned}
H(k,0,x,f) &= fx\\
H(0,j',0,f) &= H(0,j,1,f)\\
H(0,j',x',f) &= H(0,j,y,f)\\
\text{where } y &= H(0,j',x,f)\\
H(k',j',x,f) &= H(k,x,x,g)\\
\text{where } g &= H(k',j,\cdot,f)
\end{aligned}
$$

for $k, j, x \in \mathbb{N}$, $f \in \mathbb{N}'$. Let

$$
\begin{aligned}
&[k,j,x] \text{ abbreviate } (\forall f : \mathbb{N}')[H(k,j,x,f) = ((P^k R)^j f)x]\\
&[k,j,\cdot] \;\text{ abbreviate } (\forall r : \mathbb{N})[k,j,r]\\
&\qquad\qquad\vdots
\end{aligned}
$$

and so on. The four clauses of the specification give

$$
\begin{array}{lr}
(1) & [\cdot,0,\cdot]\\
(2) & [0,j,1] \Longrightarrow [0,j',0]\\
(3) & [0,j,\cdot] \wedge [0,j',x] \Longrightarrow [0,j',x']\\
(4) & [k,\cdot,\cdot] \wedge [k',j,\cdot] \Longrightarrow [k',j',\cdot]
\end{array}
$$

respectively. These are verified in the usual way.

We use (1,2,3,4) to show $[\cdot,\cdot,\cdot]$ by a quadruple induction.

From (4) an induction over j gives

$$[k,\cdot,\cdot] \wedge [k',0,\cdot] \Longrightarrow [k',\cdot,\cdot]$$

and hence

$$[k,\cdot,\cdot] \Longrightarrow [k',\cdot,\cdot]$$

follows by (1). Using this an induction over k gives

$$[0,\cdot,\cdot] \Longrightarrow [\cdot,\cdot,\cdot]$$

so it suffices to verify $[0.\cdot,\cdot]$. But this is just

$$(\forall f : \mathbb{N}', j, x : \mathbb{N})[H(0,j,x,f) = R^j f)x]$$

which is essentially the result of Theorem 6.7, and follows from (1, 2, 3) by a double induction over j, x.

The right-hand case is dealt with in a similar way. ■