Principles of Computation and Relative Computation

Konstantine Arkoudas

June 1994

Contents

List of Figures

Part I

Computability

Chapter 1

The big picture

1.1 The object of the game

Computability theory studies those problems that can be solved mechanically, while relative computability (or *un*computability, as it is otherwise known) studies those problems that can*not* be solved mechanically. What does it mean for a problem to be 'mechanically solvable'? Well, that's a good question, and we will in fact make it the starting point of our inquiry. It is the very same question that Post and Turing and Church and Godel and many other brilliant researchers were asking about sixty years ago. And it is an easy question to understand and formulate (though not as easy to answer). For, to some degree or another, we are all familiar with mechanical solvability, as we all know of at least a few problems that can be solved in a mechanical fashion. Take the problem of finding the sum of two integers, for example. In elementary school most of us learnt a mechanical *method* for solving this problem, namely the traditional addition procedure: first we align the two numbers vertically and then we start adding individual digits column by column, going from the right to the left, keeping track of the carries, etc. We say that a problem such as this is mechanically solvable because there is a mechanical procedure, an *algorithm*, which we can blindly use to produce the answer to *any* given instance of the problem. And we say "blindly" because a mechanical procedure does not require any intelligence whatsoever on the part of the agent who is carrying it out. That is why we call it "mechanical" after all; because it is like a *recipe*, it tells us exactly what to do at each and every step of the way, leaving nothing to chance or creativity or imagination.

In the beginning of the present century researchers in Logic and the

foundations of Mathematics began to look a little deeper into mechanical solvability. They started asking quite profound questions, such as "Is there a mechanical procedure which, given an arbitrary mathematical statement S, can tell us whether or not S is true?" It is not difficult to appreciate the foundationally paramount importance of such a question. An answer to it, whether it be affirmative or negative, is obviously bound to have tremendous philosophical ramifications with regard to the nature of mathematical truth and its accessibility to humans. But a question cannot be answered unequivocally until we define its terms with precision. The above question (and many other similar ones that were being asked at that time) is no exception. Before we can say whether or not the problem of determining mathematical truth is mechanically solvable, we must know exactly what we mean by 'mechanically solvable problem', and *a fortiori*, exactly what we mean by 'problem'. We must, as they say, **formalize** the concepts of problem and mechanical solvability. This is what the aforementioned logicians set out to do in the early 30s, and this is what we will set out to do, from our contemporary vantage point, in the first part of the essay. In chapter 2 we will define problems and in chapter 3 we will define what it means for a problem to be mechanically solvable. But first it will be a good idea to understand what formalizations are all about, what it means to formalize a concept. Doing so at the very beginning of our exposition will pay off handsomely later, when we come to discuss some very important issues in computability, such as Church's thesis, which are inextricably linked with the subject of formalizations.

1.2 Formalizations. Their importance, social context, and feasibility.

What does it mean to formalize a concept? Before we can answer this, we must be sure of what we mean by 'concept'. We will use this word as liberally as possible: almost anything that is not a **particular** will count as a concept. Thus we have the concept of chairs, for example, though no particular chair is a concept. Or the concept of number, though no particular number is a concept. Or the concept of manhood, though individual men are not concepts. Particular chairs are rather said to be **instances** of the chair concept. Likewise, a particular man, say George Washington, is an instance of the concept of man.

The **extension** of a concept is the aggregate of all its instances. For example, the extension of the chair concept is the class of all chairs; the

extension of the man concept is the class of all men, etc. Asides from extensions, concepts are also said to have **intensions**. Intensions are not as easy to explain as extensions. For our purposes here we will take the intension of a concept to be its *meaning*, i.e all those properties and characteristics that apply exclusively to the instances of the concept. The intension of the chair concept, for example, involves all and only those things that we normally associate with chairs: being an individual piece of furniture, having the ability to support buttocks, etc. The intension of the concept "president of the USA" involves residing in the White House, serving a 4-year term, being the commander-in-chief of the army, being involved in scandals, etc. Note that a concept might have an intension without having an extension. Take the concept of unicorns, for example, or the concept of honest politicians. Both of these have clear meanings (intensions), despite the fact that they have no instances. What is more relevant to our discussion, two concepts might have the same extension but different intensions. The classic example here (given by Quine) is the concept of cordates (creatures that have hearts) and that of renates (creatures that have kidneys): despite their intensional difference, these two concepts are co-extensive, i.e the instances of the first are all and only the instances of the second. We will see that in some cases the effect of an accurate mathematical definition is to replace an intensionally vague concept A with a concept B that has the same extension as A but a more formal intension.

A lot of concepts are **fuzzy**, i.e it is not clear exactly what their extensions are. Take the concept of young persons for example. Now there are some people whom we would definitely call young without a single moment's hesitation (newborns, for example). And, alas, there are those people of whom we would definitely say that they are *not* young (my 88-year-old grandfather is one of them). But then there is also a great number of people whom we would feel equally uncomfortable in calling either young or not young. These people are not clear instances of the "young person" concept, but nor are they clear non-instances. They are "border-line" cases.

In contradistinction to fuzzy concepts we have **sharp** concepts. A concept A is sharp iff it has a precisely delimited extension, i.e iff any given object is bound to be either a definitive instance of A or a definitive non-instance. Consider the concept of 20-year-old persons, for example. At any one time every person is either 20 years old or not— there are no border-line cases. Furhermore, we can distinguish betwen two kinds of sharp concepts: those that are **evidently sharp** and those that are not. We will call a concept A evidently sharp iff the assumption that A has a border-line instance automatically engenders a contradiction. In other words, evidently

sharp concepts are provably sharp. Most sharp concepts fall in this category. Consider "6 ft. tall", "born in August", "divisor of 38", etc. We might say that asides from a sharp extension, an evidently sharp concept also has a *sharp intension*. As was already said, there are some sharp concepts that are not evidently sharp. Most foundational concepts in the mathematical sciences are initially non-evidently sharp, until they are rigorously defined, i.e until they are formalized (more on this shortly). Take continuity for example. Before Cauchy defined continuous functions with precision, the concept of such functions was non-evidently sharp. On the one hand it was clear that the concept must have a sharp extension, as it was intuitively obvious (from geometric considerations) that any function will necessarily be either continuous or discontinuous. On the other hand, one could not *prove* that. Hence, the concept was what we have here called "non-evidently sharp". As another example, consider Tarskian definitions of truth in first-order theories, say in arithmetic. Prior to such definitions, the concept of "true arithmetic statements" was non-evidently sharp. We might say that concepts of this kind are "intensionally fuzzy", despite the fact that they have sharp extensions. By contrast, evidently sharp concepts, as we have already remarked, are both extensionally and intensionally sharp.

To formalize a concept is essentially to *make it* evidently sharp. This is usually done by *defining* the concept in a precise manner. A definition is a statement of the form "A is B"[1] where A is the concept [2] being defined (the **definiendum**, plural **definienda**) and B is the concept in terms of which we are defining A (B is called the **definiens**, plural **definientia**). We will say that a definition is **rigorous** iff the definiens is an evidently sharp concept. Consider, for example, the following definition:
A self-evident geometric proposition is one whose truth can be easily and distinctly conceived by the human mind.
This definition is *not* rigorous because the definiens is not even sharp, let alone evidently sharp. Its fuzziness is clearly evinced in the vague, imprecise

[1] A lot of definitions, especially those which we will later call predicate definitions, are given in the "iff" form, i.e they list a set of necessary and sufficient conditions for membership in the extension of the definiendum. They can all be straight-forwardly rephrased, however, as statements of the form "A is B".

[2] It might be argued that in a definition "A is B" the definiendum is not a concept but rather the term that we use ('A') to refer to the concept. Similarly, we have been talking about concepts having meaning, against which it might be said that strictly speaking the bearer of meaning is not the concept itself but rather the term (the linguistic token) that we use to signify the concept. We will not pay much attention to such fine distinctions here, in order to avoid clumsy circumlocutions, but the reader should be at least minimally aware of them.

terms it contains ('easily', 'mind', 'conceived', etc.). A definition that comes closer to being rigorous is the following:
A function f from the real numbers to the real numbers is everywhere continuous iff the graph of f has no breaks or gaps.
This definition fares better in rigor than the preceding one because at least intuitively we can see that the definiens is a sharp concept. It is not *provably inconsistent*, however, to assume the existence of a fuzzy (i.e border-line) instance[3], despite the fact that such an assumption is psychologically in full contradiction to our intuition. The problem here is that the definiens is not what we have called evidently sharp— its intension is not formal enough. Consequently, the definition is not rigorous. Finally, consider the following definition:
A self-evident geometric proposition is one that can be inferred from Euclid's axioms, using a fixed subset of the inference rules of first-order Logic, in no more than 20 steps.

This *is* a rigorous definition because the definiens is evidently sharp. Clearly, any geometric proposition will either be an instance of the definiens (it will be deducible from Euclid's axioms in no more than 20 steps) or it will not— it is trivially evident that there is nothing in between. Hence the rigor. Finally, we put it all together and explicate the activity of formalizing as follows:

A formalization is a rigorous definition of a fuzzy or non-evidently sharp concept.

In other words, to formalize a concept is to define it rigorously. Of course the concept being defined must be either fuzzy or non-evidently sharp, for otherwise it must be evidently sharp, in which case there is obviously no need to formalize it. Also, note that we are not talking here about the quality of definitions, only about their rigor. Defining a computer as "an IBM PS/2 personal computer with a 286 processor, manufactured in 1991" is a horrible formalization of the computer concept, but it is a formalization nevertheless. It is obvious that there are good definitions and bad definitions and telling the good ones from the bad ones is perhaps the most important key to doing good science. We will take up this very important topic in the next section. For now let us just say that we require *a lot* of a definition before we count it as a *good* formalization. But to count it simply as a formalization, all we require of it is an evidently sharp definiens.

[3]Unless, of course, we have fixed precise technical meanings for 'break' and 'gap' which do make such an assumption logically inconsistent; here we take it that this is not the case, i.e we assume a common informal understanding of these terms.

The natural question at this point is to ask what is the value of formalizations. What good are they? In purely empirical sciences such as botanology or geology, the value of rigorous definitions lies mainly in:

1. **Organizing knowledge in a systematic way by removing ambiguity and dividing up cocepts in neat taxonomies.** By giving meticulous *per genus et differentia* definitions of different types of insects, for example, we systematize our knowledge of them so that the recording and retrieving of known facts, and even the discovery of new ones, becomes easier.

2. **Assuming legislative power for instances that were borderline cases prior to the formalization.** The contemporary scientific definition of 'fish', for example, does not include whales. Whatever ambiguity might have existed previously as to whether or not whales are fish has now been removed by stipulation.

3. **Facilitating the formulation of laws.** A major goal of empirical definitions is to help the formulation and upholding of physical laws. By defining fish so as to exclude whales, for example, we immediately validate the statement "All fish are cold-blooded", to which whales would constitute a counter-instance were they to be counted as fish.

All of the above benefits also hold good in Logic and Mathematics. In fact these benefits are often realized more dramatically in mathematical fields because formalization there is more urgent, and sometimes more potent, too. Because knowledge in mathematics is arrived at by means of deductive rather than inductive inference, it cannot even begin in the absence of rigorous definitions. If a mathematical concept has not been defined perfectly unambiguously, there is *nothing* we can prove or disprove about it (save trivial tautologies and contradictions). To see this, suppose we want to prove that all instcances of a certain concept A have a certain property P. To prove something like that, we would probably have to consider an arbitrary instance i of the concept, adduce a chain of inferences leading to the conclusion that i has P, and then use universal generalization to establish that all instances of A have P. Now the point is that at some step or other in the above chain we *must* bring in and rely on the defininig characteristics of i (for if we can give the proof without needing to know anything about the nature of i as an instance of A, then either we have proved a tautology or we have not proved anything). Obviously, then, if precise defining characteristics are not available, such proofs are not possible. The upshot

is that if we are to count as mathematical knowledge only what has been proved, as is indeed customary, then we cannot really know anything about an unformalized concept. Only after the concept has been given a clearly sharp extension can we go ahead and incorporate it in a formal *theory*. This is why we say that a formalized concept is **theoretical**, whereas the time period prior to the formalization is known as the concept's **pre-theoretical** era.

Moreover, formalization in mathematics can often pay greater epistemic dividends than it does in the empirical sciences. Whenever it is substantial, this disparity is due to the fact that in the pre-theoretical era of an empirical concept we are usually acquainted with as many instances of the concept as we are with non-instances. Before we defined fish with precision, for example, we were as thoroughly familiar with fish as we were with non-fish such as birds and elephants and humans. The main immediate benefit of the formalization was the stipulative elimination of ambiguity—we could now classify something as a fish or as a non-fish with certainty. The situation is similar in mathematics whenever we have a strong pre-theoretical intuition about a concept. In the case of function continuity, for example, prior to formalization we were acquainted with many clear instances of the concept (indeed, most common functions were "obviously" continuous), but we were also acquainted with many clear *non*-instances of it, such as functions with "breaks" or "gaps" in their graphs. It was "obvious", for example, that functions like $\lfloor x \rfloor$ or $\lceil x \rceil$ (ceiling and floor) are discontinuous, though there was no way to prove that formally. So in this case, too, the main *immediate* benefit of the formalization was the ability to *prove* that functions like polynomials are continuous whereas functions like $\lceil x \rceil$ are not (of course the long-run epistemic benefits were much more profound). However, there are other cases in mathematics where our pre-theoretical acquintance with a concept is restricted to one side of the fence only, the affirmative side. In cases of this sort the nature of the concept is such that its informal version can only reveal to us instances of itself, not counter-instances. In such cases, the rigor that is introduced by a successfull formalization enables us not only to prove that all of the instances we had previously observed are indeed instances, but also to deductively *discover* non-instances— a task that would have been practically impossible before the formalization. This was in fact the case with the concept of mechanical solvability. Before Turing's formalization we were acquainted with many problems that were "obviously" amenable to mechanical solution: finding greatest common divisors, differentiating polynomials, etc. However, we had no clear examples of problems that were *not* mechanically solvable, at least in the sense in which we had

clear examples of non-continuous functions. Turing's definition enabled us to discover an entire class of mechanically *un*solvable problems and, indeed, to ultimately find out that the vast majority of problems are so unsolvable. This is the kind of immediate knowledge boost that is not likely to occur in the physical sciences, as empirical concepts usually have fairly clear complements (negations).

We have divided the lifetime of a concept into two distinct periods, pre-theoretical and theoretical, the dividing line between the two being formalization. The transition from the former period to the latter is of course a social phenomenon and it is usually gradual and intellectually arduous— at least for important foundational concepts. For purposes of illumination, we may distinguish three stages in a concept's pre-theoretical era:

1. The stage of **initial acquaintance**,
2. The stage of **increasing familiarity**, and
3. The stage of **abstraction**.

The third stage is attained from the second via induction, while the second seems to be derived from the first by means of purely psychological processes related to memory. The temporal dimensions of these stages usually vary significantly from concept to concept, but it is interesting to note that as time has gone by stages (1) and (2) have typically decreased in duration to the point where nowdays our acquintance with many new concepts begins almost immediately at the stage of abstraction. To a large extent this is due to the increasing fluency of mathematicians with formal languages, in combination with the important changes that took place during the last hundred years in the way in which mathematics is taught and practiced— changes that have increasingly emphasized and valued the transition from the concrete to the general and abstract. Both of these trends have been greatly linked with and abetted by the explosive co-eval development of Mathematical Logic; but these are historic observations and need not be elaborated further. Instead, let us give an example by applying our little theory to the function concept.

In this case the stage of initial acquaintance is that in which we (humans) first observe that some particular thing A, a physical quantity of some sort, for example, is curiously *dependent upon* another physical quantity B, in the sense that if we vary B, A will also change according to some kind of pattern. If I am near a big burning oven, for instance, it is obvious that how hot I will feel as the result of the oven, i.e how high the temperature of

my body will be, *depends on* how close I stand to the oven, or, equivalently, on how great the distance is between my body and the oven. The closer I am to the oven, the hotter I will be. The further away I get from it, the less it will affect me. Borrowing from the every-day vocabulary of our language, we may say that T, my temperature, is *a function of* D, the distance between my body and the oven. We may then proceed to quantify T and D in accordance with some metric system (perhaps speak of T in Kelvin degrees and of D in feet), introduce the Cartesian co-ordinate system by putting D on the x-axis and T on the y-axis, take care of boundary cases by reasonably stipulating that for $D = 0$ our body temperature *is* the temperature of the oven (and we are quite properly roasted) while for $D = \infty$ T is (presumambly) the normal body temperature, and there we have it—the rudiments of a perfectly reasonable mathematical model of our physical situation. We have just become acquainted with the first concrete instance of "function-hood".

Of course this is a quite over-simplified account of what happens in actuality, but I think it draws the right picture (albeit a little roughly) of how we first come to conceive of a concept. We can draw an analogy between this stage of our acquaintance with an abstract concept and the situation in which several pre-historic people on a misty hill top are looking down at the valley at the first horse they ever saw. At first they are unsure of its shape, texture, intentions, and overall nature, mainly because of the mist. Then they descend the hill and approach the horse until they are right next to it, examining it with all the scrutiny and marvel of a first experience, until they become more familiarized with it, take it for a ride, and become generally accustomed to its "horseness".

Returning to the function example, now that we have made our acquintance with one of them, we see functions everywhere we look. Our expenditures are functions of our incomes, forces on material objects are functions of masses and accelerations, our hunger is a function of the time that has elapsed since our last meal, and we now become bold enough to suggest that the temperature around the oven is really a function of *three* variables, the x, y, and z spatial co-ordinates of any one point in the surrounding space. We are now well into the stage of increasing familiarity, and for the purpose of facilitating communication and discourse, we introduce the term *function*, which thus acquires a standard, though not yet *precise* technical meaning. In our horse analogy, the corresponding stage would be the one in which our pre-historic people encounter more and more horses, become increasingly familiar with them, and the term 'horse' becomes well entrenched into the every-day vocabulary of their native language.

Having experienced a sufficient number of concrete instances of "function-hood" we now enter the stage of abstraction in which we reason inductively and hypothesize the existence of an abstract concept of which all the witnessed instances partake. At this point we have a solid pre-theoretical understanding of the concept, and although we know that the term we have coined must signify *something*, we are not sure exactly what that something is. So the effort at this point is directed towards finding a crisp extension that will somehow manage to capture the intension that has developed. Generally speaking, the denoument of the stage of abstraction is a successfull formalization. In our case that would be the definition of a function as a set of ordered pairs, no two of which have identical first and distinct second elements. In the horse analogy, this would be the stage in which we arrive at a precise definition of 'horse' that successfully captures the animal's zoological properties.

1.3 Definitional accuracy

As was already pointed out, there are good definitions and there are bad definitions— that much is obvious. The quality of definitions, or rather their *accuracy*, is an extremely important issue in all sciences. A really good definition of a key concept can prove to be a tremendously powerful catalyst for scientific progress, making previously nebulous ideas fall into place and opening up new intellectual avenues and possibilities. By contrast, a defective (inaccurate) definition can pose a grave epistemic hazard. Not only can such a definition steer us away from theoretical truth, it can actually lead us to believe what are in fact false propositions (namely, its logical consequences). In this section we will try to find out what makes a definition accurate. First we will divide definitions into four distinct classes. For three of those classes we will be able to formulate an explicit criterion of accuracy, which will be given in the form of necessary and sufficient conditions. That criterion will not constitute a "hard and fast" rule that will tell us whether or not an arbitrary definition is accurate. Its utility will be more theoretical, in the sense that it will give us a general idea of the conditions that must be satisfied by an acceptable definition. But it will not be entirely devoid of practical value, either, although that value will be of a somewhat negative character. In particular, our criterion will tell us exactly what would have to happen in order to reject a definition as inaccurate (and hence, indirectly, what should *not* happen if we are to accept the definition). But, again, it will not give us explicit instructions on how to establish a definition as

accurate. Indeed, it will be seen that only the empirical test of time and experience can ultimately establish such accuracy. Finally, for definitions of the fourth kind, which we will call "mathematical entity definitions", we will see that the issue of accuracy is considerably more complex and that no explicit criterion can be given. What we will try to achieve for such definitions is an intuitive understanding first of their nature and then of the general factors that need to be considered in assessing their accuracy.

We now proceed to introduce our distinctions. First we will distinguish between **entity definitions** and **predicate definitions**. A predicate definition will be one which defines a *property* P that some objects might or might not have. Predicate definitions are given when we have a certain class of objects U (the "universe of discourse") and we want to make clear which objects in U have P and which do not. If the definition is successfull, then the extension of the definiens will comprise all and only those things in U that clearly have P. An example of a predicate definition is that of the 'real functions of one real argument that are *continuous* over [0,1]'. Here the universe of discourse U is the class of all functions from the reals to the reals, while the property being defined is 'continuity over [0,1]'. Other examples of predicate definitions are those of 'prime integer', 'educated American', 'countable set', 'large city', 'valid formula', etc. On the other hand, entity definitions define *objects*, not properties of objects. An entity definition purports to hypostasize a certain concept, i.e to tell us *what* its instances are. One can think of such definitions as lessons in ontology. The definition of 'function' is an example. It tells us *what* a function is (namely, a set of ordered pairs such that ...). The definitions of 'number', 'man', 'sequence', 'horse', 'formula', etc., are additional examples. Loosely, then, we might say that property definitions define adjectives, while entity definitions define nouns. Finally, we claim that our distinction divides definitions into two mutually exclusive and jointly exhaustive classes, i.e any given definition will either be of the predicate kind or of the entity kind. For even definitions which at first glance do not seem to fit either category can be equivalently reformulated as clear predicate or entity definitions. The presently accepted definition of heat, for example, might be recast so as to define heat as a unary property of arbitrary collections of molecules.

We are now ready to introduce our second distinction. In what follows we will be making frequent use of the term 'mathematical object'. The meaning of this term should be obvious. Things like sets, functions, sequences, ordered pairs, relations, cartesian products, numbers, etc., will count as mathematical objects; things like chairs, houses, stones, etc., will not. We could restrict the term even further by having it denote only sets in Zermelo's

cumulative universe, but it will be sufficient to rely simply on our intuitive understanding of it. Having said that, we proceed to explain the distinction between **mathematical definitions** and **empirical definitions**. We will call a definition mathematical iff the instances of the definiens are mathematical objects, *unless* those objects are n-tuples of non-mathematical objects and the definiendum is an n-ary predicate, where $n > 1$. If this exception condition obtains, the definition will not count as mathematical. Any definition that is not mathematical will be called empirical. Hence, the definition of prime integers is mathematical, since the instances of the definiens are mathematical objects (integer numbers), whereas the definition of horses is obviously non-mathematical and thus empirical. Note that the exception condition in our specification of mathematical definitions was included only in order to accomodate definitions of empirical polyadic relations. For example, in the definition of "City X is the capital of country Y", the extension of the definiens consists of ordered pairs, which are mathematical objects. The elements of these ordered pairs, however, are *not themselves* mathematical objects— they are cities and countries. Thus our exception clause makes sure that this definition is not counted as mathematical, which is obviously what we should want. However, since our discussion will *not* focus on polyadic predicates, the reader can for all practical purposes ignore the qualification and think of a definition as mathematical iff the extension of the definiens consists of mathematical objects. Hence, according to the terminology we have introduced so far, any given definition will be one of the four following kinds:

- An empirical entity definition (A *horse* is ...)
- An empirical predicate definition (A *wild horse* is ...)
- A mathematical entity definition (A *function* is ...)
- A mathematical predicate definition (A *continuous function* is ...)

We now go ahead with our quest for a criterion of definitional accuracy.

As a first attempt, one might be inclined to propose extensional identity as the sought-after criterion. That is, we might say that a definition is accurate iff every instance of the definiendum is also an instance of the definiens and vice-versa; in other words iff the definiendum and the definiens have identical extensions. That seems to make sense. We would think that a definition of 'horse' is accurate iff the definiens included all and only those things that we call 'horses'. However, when one thinks about it a little more

it becomes apparent that extensional identity cannot be what we are looking for. There are two main reasons for that.

Problem #1: First of all, it might be that the definiendum is a fuzzy concept. In that case strict extensional identity with the definiens is impossible, as no fuzzy concept can ever coincide exactly with a sharp concept— it wouldn't be fuzzy if it did. Yet there are several definitions of fuzzy concepts in the sciences and most of them are rightly regarded to be quite accurate, despite the lack (indeed, impossibility) of extensional identity.

Problem #2: A second and more important reason why extensionality fails has to do with mathematical entity definitions. As was already said, in definitions of this kind the instances of the definiens are mathematical objects. But the instances of the definiendum are not. For, if they were mathematical objects, then the definition would either say something false or something of the form a = a, i.e something trivially true. Take Von Neumann's definition of (natural) numbers, for example. The instances of the definiens are the finite ordinals $\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \ldots.$— perfectly crisp objects with crystal clear identity conditions. But what are the instances of the definiendum? What *is* a number after all? The answer is: nobody really knows[4]. Indeed, from an intuitive viewpoint it is far from clear that numbers are sets, let alone sets in the cumulative universe and, moreover, those particular sets in that universe that are singled out by Von Neumann's definiens. Extensional identity, therefore, is highly unlikely to obtain in such cases. Of course against this one might argue that the apparent ontological dissimilarity between the definiendum and the definiens of a mathematical entity definition is just that: apparent; and that careful analysis will inevitably show that the instances of the definiendum are really "nothing but" the instances of the definiens. Although nobody really knows what numbers are, for example, and although they apparently do not seem to be sets of anything, a thorough analysis will reveal that they are really nothing but the finite ordinals $\emptyset, \{\emptyset\}, \ldots.$ To this, however, the fatal reply can be made that the same definiendum can be accurately defined by two different definitions, the definientia of which have non-identical (indeed, often disjoint) extensions. But if strict extensional identity were a correct criterion of definitional accuracy, then it would be obviously impossible for more than one mathematical entity definition to be accurate. Suppose, for example, that, wishing to uphold extensionality, one asserts that Von Neumann's definition is accurate *because*:

[4]This point is elaborated further in section 2.2.

(**I**) "The natural numbers *are* exactly the finite ordinals."
It would then follow that a different definition of numbers ought to be *a priori* ruled out as inaccurate because the definiens would be extensionally different from Von Neumann's definiens and hence, by (**I**), extensionally different from the definiendum. But there are several alternative set-theoretic ways to define the natural numbers with accuracy, and the same goes for many other mathematical entity definitions (e.g for ordered pairs). Therefore, the existence of distinct but equally accurate mathematical definitions of one and the same class of entities establishes the incorrectness of the extensionality criterion.

We might try to overcome these difficulties by suggesting the following. **CRITERION OF DEFINITIONAL ACCURACY:** *A definition is accurate iff (i) every clear instance of the definiendum is an instance of the definiens, and (ii) no instance of the definiens is a clear non-instance of the definiendum.*

It is easy to see that this circumvents problem 1. It essentially grants definitions the leeway that is necessary for the formalization of non-sharp concepts (by giving them sufficient legislative authority in dealing with borderline cases) while at the same time insisting that they do not contradict our pre-theoretical understanding of the non-border-line parts of the concept. Let us call a definition **under-inclusive** if it violates clause (i) of the above criterion, and **over-inclusive** if it violates clause (ii). Then a second (and perhaps more intuitive) way to express the criterion is to say that an accurate definition is one which is neither under- nor over-inclusive. Suppose, for example, that we defined "self-evident geometric proposition" as "one which follows from Euclid's axioms in no more than 10 billion steps" (using a fixed set of rules of inference). Then although this definition is not under-inclusive, it is over-inclusive. For, consider a proposition that follows from Euclid's axioms in 5 billion steps but no less. Such a proposition is obviously an instance of the definiens, since 5 billion steps are fewer than 10 billion steps. But it is also a clear non-instance of the definiendum, for we would never say of such a proposition that it is "self-evident". Of course under- and over-inclusiviness are not mutually exclusive defects. An example of a definition which suffers from both is one which defines a continuous fuction f of one real argument as one for which $f(0) = 0$. Clearly, there are uncountably many discontinuous functions which are included in this definiens and uncountably many continuous ones which are excluded.

It should be fairly clear by now that our criterion works for (1) empirical entity, (2) empirical predicate, and (3) mathematical predicate definitions, regardless of whether or not the definiendum is sharp (indeed, if the

definiendum is (non-evidently) sharp, then our criterion collapses to strict extensional identity). And it works fine because in definitions of any one of these three kinds we go from apples to apples, or from oranges to oranges, so to speak. What we have in the extension of the definiens is the *exact* same type of stuff that we have in the definiendum, so it is legitimate to require that every (clear) instance of the latter should *itself be an actual member* of the extension of the former; and that no instance of the definiens should itself be an actual member of the (clear) complement of the extension of the definiendum. For example, in an accurate empirical entity definition of 'horse', instances of the definiendum are themselves actual instances of the definiens and vice versa— we have horses in both. The same goes for an empirical predicate definition like that of an "educated American"— clear instances of the definiendum should be actual instances of the definiens. Likewise with mathematical predicate definitions like that of continuous functions: all those functions that we deem intuitively continuous should be actual instances of the definiens and conversely. Mathematical entity definitions, however, are as much of a problem to our criterion as they were to the extensionality criterion because in such definitions we seem to be going "from apples to oranges". For, although we know exactly *what* comprises the extension of the definiens, we have no similar knowledge as regards the extension of the definiendum. Therefore, to require that the (clear) instances of the definiendum should *themselves* be actual instances of the definiens would be both senseless and incorrect. First it would be senseless because our intuition would then rule out most mathematical entity definitions as inaccurate, as we would hesitate to concede, for example, that the ordered pair $< 8, 5 >$ *is* the Kuratowski construction $\{\{8\}, \{8, 5\}\}$. And it would also be incorrect because if we committed to the view that $< 8, 5 >$ *is* $\{\{8\}, \{8, 5\}\}$, then our criterion would automatically reject as inaccurate any other definitions of ordered pairs of numbers— which is unacceptable. It appears, then, that mathematical entity definitions need separate, individual treatment. Our approach will be to stop viewing such definitions in the same way we view definitions of the other three kinds, i.e as statements with definitive truth values dependent partly on metaphysical fact and partly on linguistic convention.

In particular, I think it is best to view a mathematical entity definition as a *proposal*. When we define a sequence of real numbers, for example, as a function from the positive integers to the reals, we are essentially *suggesting* that it is a good idea to view a sequence of real numbers as a function from the positive integers to the reals. Our definiens isolates a certain class C of mathematical objects and we propose that we start thinking of instances of

the definiendum (whatever those might *actually* be) as instances of C. We are not categorically *equating* instances of the definiendum with instances of the definiens; we are simply saying that it is pragmatically expedient to view instances of the definiens as formal extensional *counter-parts* of instances of the definiendum— whatever those latter instances might be in actuality, for we do not care much about their actual metaphysical essence any more. Indeed, a mathematical entity definition always establishes an understanding which enables us to know which instance of the definiens is the proposed formal counter-part of any given instance of the (informal) definiendum. E.g when we define sequences of reals as above, we know (by knowing the definition's rationale) that the formal counter-part of the sequence $s = [3.14, 5, e^{10}]$ is the function $f = \{< 1, 3.14 >, < 2, 5 >, < 3, e^{10} >\}$. In other words, f is the *hypostasis* that the definition proposes to attribute to the informal object s. And conversely, given an instance x of the definiens, we can tell which instance of the definiendum x is the proposed counter-part of. E.g we know that the function $\{< i, \pi^i > | i \in Z^+\}$ is the formal counter-part of the (infinite) sequence $[\pi, \pi^2, \pi^3, ...]$.

If we see mathematical entity definitions in this light, it becomes difficult to formulate a clear-cut criterion for evaluating their accuracy. Of course one thing that must definitely hold is the following:
For every clear instance i of the definiendum there must be an instance i' of the definiens (the proposed counter-part of i) which we can "think of" as i; and for every instance i' of the definiens there must be an instance i of the definiendum which we can "think of" as i'.
Of course this is considerably vague because what is really needed for every i in the definiendum is not just a counter-part i' in the definiens, but a *good* counter-part, one which we can *naturally* "think of" as i. But even though an explicit criterion of accuracy might be an impossibility for mathematical entity definitions, there are some heuristic rules of thumb that all accurate definitions of that kind should obey. Firstly, the instances of the definiens should encapsulate as much "information" about their informal counter-parts as is extensionally possible. The more information they contain the better they will characterize the objects they supposedly stand for. The main reason why the formalization of the function concept was so successfull is because as a formal, extensional object, a set of ordered pairs (with the usual uniqueness property) contains *exactly* the information that is necessary and sufficient to *fully* characterize a "function". On the other hand, if the pre-theoretical objects we are trying to define are not so "extensional" to begin with, i.e if we cannot easily define them simply by lumbing together all there is to know about them in some sort of aggregate

structure, then usually what matters is not *what* objects we propose but *how* they behave. This is the case with numbers, for example. Because numbers are not really characterized by what they contain but rather by what laws they obey, it has been possible to define them in ways that are extremely disparate from an extensional standpoint. In Russell's definition, for example, a whole number, say 2, is a huge collection (in terms of cardinality), whereas in Von Neumann's definition the same number has a scanty total of two elements, both of which are pretty small themselves ($\emptyset$ and $\{\emptyset\}$). In both definitions, however, the instances of the definiens behave "as they should": we can define on them operations such as addition and multiplication, such operations turn out to have all the usual desired properties (commutativity, associativity, etc.) and so on. And that is sufficient reason to accept both definitions, in their own contexts, as accurate.

Another principle we might use as a rule of thumb is **substitutability with preservation of truth value**. Let S be any sentence containing a term t signifying one or more instances of the definiendum; and let S' be the sentence obtained from S by replacing t with a term t' that signifies t's counter-part instance(s) in the definiens. The principle asserts that if the definition is accurate then S and S' should have identical truth values, i.e either they should both be true or they should both be false. Of course the truth or falsity of the original sentence S should be entirely independent of the accuracy of the formalization. Otherwise we will have no way of knowing whether the substitution preserved the truth value of S. In the customary definiton of functions, for example, we would not apply the test to a sentence like "Functions are not sets". For, although the sentence we would obtain from the substitution would be clearly false, we have no way of knowing whether the original sentence is also false *unless* we presuppose the definition's accuracy— which, however, is what the substitution was intended to test in the first place.

Truth-value-preserving substitutability has often been proposed as a general criterion of accuracy for *all* types of definitions. Unfortunately, troublesome sentences of the kind we discussed above lead to various awkward qualifications and circumlocutions which ultimately render the principle unfit for the role of a comprehensive criterion. Informally, however, substitutability can obviously lend some useful insight concerning the accuracy of a particular definition.

Regrettably, I doubt whether the issue of accuracy of mathematical entity definitons can be analyzed any further. Whether such a definition is a "good" proposal or not is a question that must ultimately be examined on an individual basis. The guiding principle will always be a combination of

intuition and pragmatic considerations. These two factors are usually interwoven anyway. A mathematical entity definition which does not appeal to the intuition is not very likely to "work" and yield the "right results".

Thus we see that definitional accuracy, at least for mathematical entity definitions, is a wholly empirical issue. But the same is actually the case for the other three types of definitions: empirical entity, empirical predicate, and mathematical predicate. For we agreed to call a definition of these kinds accurate iff every clear instance of the definiendum is an instance of the definiens and every instance of the definiens is an instance of the definiendum. But this is a universally quantified empirical statement. Universally quantified because it talks about *all* instances of the definiendum and the definiens, and empirical because of the word "clear", whose meaning is inextricably linked with human mental processes. Therefore, to claim that a particular definition is accurate is akin to making a claim like
(II) "All storks have red legs".
Such statements can never be completely confirmed. Only their negations can. We can establish the falsity of **(II)** conclusively, once and for all, simply by finding a stork which does not have red legs. We could then definitively reject **(II)** as untrue. But we can never establish **(II)** as true. The best we can do is amass increasing amounts of evidence in its favor— the more storks with red legs we encounter the more confident we become in the truth of our claim. The same goes for claims of definitional accuracy. We can never *prove* that a definition is accurate. We can simply claim such accuracy with varying degrees of conviction depending on the amount of the available evidence (i.e the number of clear instances of the definiendum that have been seen to be instances of the definiens, or other such empirical considerations). But because the evidence is inductive, the possibility of counter-examples will always be open, if not in practice then at least in principle.

We end this section with a final word of caution concerning mathematical entity definitions. Most such definitions, even those that are highly successfull, are bound to be "slightly over-inclusive". I am not talking about vicious over-inclusiveness here, where the definiens has instances which cannot be thought of as *any* instances of the definiendum. If a definition suffers from *that* type of over-inclusiveness then it is outright inaccurate and that is the end of the story. The over-inclusiveness I am talking about is of a much more benign nature: it occurs whenever the definiens includes instances which we do not really "care about", instances which, while not flagrant non-instances of the definiendum, are nontheless entirely devoid of intentional content. One might call this "over-inclusiveness up to loss of intentionality". The definition of functions, for example, compels us to think

of a set like $A = \{< 2, 10^{89.5} >\}$ as a "function" from the reals to the reals. However, unlike sets such as $B = \{< x, ce^{kx} > | x \in R, c, k \text{ constants}\}$ or $C = \{< x, sinx > | \ x \in R\}$, which represent "genuine" functions with considerable intentional significance (B models natural exponential processes, C is a useful physical quantity), "function" A is a mere side effect of our formalization— an innocuous one, but a side effect nevertheless. By the same token, when we define an "arithmetical statement" as a well-formed formula built up from the usual first-order symbols, we inevitably include various unorthodox statements that no-one would ever care to make (e.g $(\exists x)\neg(x = x) \vee (0 > 10001018)$). Likewise, when we later come to define algorithms as Turing machines we will see that some machines act so silly that we are inclined to say that they are not really executing any algorithm at all; for we usually think of an algorithm as a coherent body of meaningful instructions whose purpose is to carry out a sensible task. But, again, although this situation occurs very frequently[5] in mathematical entity definitons, and occasionally even in mathematical predicate definitions,[6] it does not pose a serious problem. After all, a "peculiar" arithmetical statement is nevertheless an arithmetical statement, a "useless" algorithm is an algorithm, and an "unnatural" function is a function. And what our formalization of, say arithmetical statements, purports to define are arithmetical statements in general, not "interesting" arithmetical statements (the latter would probably be impossible to formalize anyway, as they do not make for a sharp concept).

Unfortunately this issue is sometimes overlooked when we start to utilize our mathematical entity definitons and derive their logical consequences. In particular, oftentimes we derive existence theorems asserting that "there is" this or that formal entity with this or that remarkable property. But, even when the proof is non-constructive, we rarely ever take the time to consider whether the object we have proved to exist has any intentional significance or whether it was extraneously picked up by the definiens of our mathematical entity definition. Usually we simple presume that the object at hand is the counter-part of a clear and significant instance of the definiendum, even when we have no justification for doing so. However, if our presumption happens to be wrong, then the result we have proved is really much more of a

[5]It is avoided only when we define each instance of the definiendum individually— that way we obviously cannot go wrong. An example of this is the definition of natural numbers. Such situations are rather rare of course, as they presuppose a (non-evidently) sharp definiendum with a fairly "small" size.

[6]Consider Weirstrass's construction of a function which is everywhere continuous and nowhere differentiable.

side effect of the particular way in which we chose to define our terms rather than an illuminating truth about the pre-theoretical entities of the definiendum. Indeed, many "startling" modern results of Mathematical Logic and Theoretical Computer Science are existence theorems asserting that there is some formal object or another having some "astonishing" property. Godel's incompleteness theorem is of this form. Several "surprising" results in abstract complexity theory are similar. Consider Blum's speed-up theorem, asserting the existence of a computable number-theoretic function[7] which has no optimal algorithm (in any sense of "optimal"). As long as we use the word "function" in phrasing the theorem, all is fine. But if we describe the speed-up theorem as asserting the existence of a (recursive) *problem* having no optimal algorithm, then we are going beyond the technical: we are *forcing* the result on the pre-theoretical objects that number-theoretic functions are supposed to model, namely, mathematical problems. First of all, doing so presupposes the accuracy of the identification of problems and number-theoretic functions. We will eventually see that there is indeed good evidence for that identification. Secondly and more importantly, however, it presupposes that *that particular function* which is proven to exist represents a "genuine" problem. But since we do not in fact know this, it is as unfair to state the theorem using the term "problem" without qualification as it would be to state it in the following way :
"There exists an immensely bizarre, useless, and irrelevant problem that has no optimal algorithm".
My purpose here, of course, is not to disparage technical existence theorems like the above (the derivations of most of them are marvels of ingenuinity anyway), but rather to stress my humble opinion that we should see them exactly as that: technical results. We should proceed from the formal and restricted to the pre-theoretical and important-sounding only with caution and on the basis of solid evidence. Otherwise the leap is unjustified.

[7] A number-theoretic function is a function $f : N^k \longrightarrow N$ for some $k > 0$.

Chapter 2

Formalizing problems

2.1 Concrete questions and general problems. First definition of general problems.

Depending on the context, the term 'problem' can have any one of numerous different meanings. Here we will try to formalize the notion of a logical or mathematical problem,[1] so we will not be concerned with family problems, car problems, love problems, etc. Now within Logic and Mathematics we will distinguish between **concrete questions** and **general problems**. A concrete question is a question that asks something about one or more *particular* mathematical objects, and whose answer is also a particular mathematical object. An example of such a question is:
(Q1) Is the integer 23 a prime number?
This poses a question about a specific mathematical object, the integer 23, and the answer is also a specific mathematical object: the number 1, signifying a "yes"[2]. Here are some additional examples of concrete mathematical questions:

(Q2) What is the (indefinite) integral of the polynomial function $P(x) = 3x^5 + 7x^2 - 18$*?*
(Q3) What is the gcd ("greatest common divisor") of 12 and 30?
(Q4) Is the formula $(p \wedge \neg q) \vee (r \wedge \neg p)$ *satisfiable?*

[1] This should be taken to include problems in Computer Science, Mathematical Economics, Theoretical Mechanics, etc., i.e all types of problems involving mathematical objects and logical reasoning.

[2] We will consistently use 1 for "yes" and 0 for "no". So had the question been "Is 16 prime?", the answer would have been 0.

(Q5) Are the two graphs depicted in fig. 2.1 isomorphic?
(Q6) What is the 48^{th} *digit in the decimal expansion of* π*?*
(Q7) Do the two Pascal functions in fig. 2.2 do the same job, i.e do they compute the same function?
(Q8) What is the 2^{nd} *derivative of the polynomial* $4x^3 - 5x^2 + 6$*?*

Figure 2.1: A concrete question: are these two graphs isomorphic?

Each of the above questions asks something about one or more particular mathematical object(s). *Q2* asks something about a particular mathematical function; *Q3* about two particular integers (12 and 30); *Q4* about a particular (syntactic) object, the formula $(p \wedge \neg q) \vee (r \wedge \neg p)$; and so on. And the answer to each question is also some fixed mathematical object.

Carrying our analysis a step further we might say that a concrete question comprises one or more **input objects** (these are the *given* pieces of information) and one **output object** (representing the *answer* to the question). For example, question #8 has two input objects: the integer 2 (signifying the ordinal 2^{nd}) and the polynomial $4x^3 - 5x^2 + 6$. The output object is the polynomial $12x^2 - 10x$. Each input object belongs to a certain mathematical *domain* (i.e is of a particular mathematical "type") and the domains of all the input objects will be collectively referred to as the **input domains** of the question. Question #1, for example, has one input domain: the positive integers. The input object to question #2 is a polynomial function, therefore the input domain is the class of all polynomial functions. Question #8 has two input domains I_1 and I_2, where I_1 is the set of all positive integers, and I_2 is the set of all polynomial functions. And

```
FUNCTION F1(x : INTEGER): INTEGER;
BEGIN
   IF x > 0 THEN
      F1 := x * F1(x - 1);
   ELSE
      F1 := 1;
END;
```

```
FUNCTION F2(x : INTEGER): INTEGER;
VAR
 a,b: INTEGER;
 BEGIN
    IF x > 0 THEN BEGIN
       b := 1;
       FOR a := 1 TO x DO
             b := b * a;
       F2 := b;
    END
    ELSE
       F2 := 1;
END;
```

Figure 2.2: Do these two Pascal subroutines compute the same function?

so on. By the same token, every concrete question has its own **answer domain**. If the question is of the "yes/no" type, like questions #1 or #4, then the answer domain is the set $\{0, 1\}$. The answer domain of *Q6* is the set $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \}$; while that of *Q2* is the class of all polynomial functions. Finally, let us say that two concrete questions are of **the same kind** if their phrasings are identical, except possibly for the terms denoting their input objects. For example, the question "Is 17 prime?" is of the same kind as "Is 1593 prime?", but *not* of the same kind as "Is 29 a perfect square?".

Now, informally, a **general problem** P is what one gets from a concrete question q by the process of abstraction. In particular, if in the phrasing of q we replace the terms used to denote the input objects with arbitrary *variable* terms ranging over the corresponding input domains, then we get a general problem. For example, "Is an arbitrary positive integer x prime?" is the general problem we obtain by abstracting away from concrete question #1. Likewise, the general question we get from *Q2* is: "What is the (indefinite) integral of an arbitrary polynomial function $P(x)$?". And so forth. A more common way to state a general problem is to use a phrase of the form *"Given ..., what is ....?"*. This way more emphasis is put on the distinction between what is given (the input object(s)) and what is sought (the answer object). Adopting this phraseology, the general problem obtained from *Q1* can be posed as: *"Given an arbitrary positive integer x, is x prime?"*. We use this style to list all the general problems arising from questions *Q1* – *Q8*:

(P1) Given an arbitrary positive integer x, is x prime?
(P2) Given an arbitrary polynomial P(x), what is the indefinite integral of P(x)?
(P3) Given two arbitrary integers x and y, what is the gcd of x and y?
(P4) Given an arbitrary formula p of propositional logic, is p satisfiable?
(P5) Given two arbitrary graphs G1 and G2, are they isomorphic?
(P6) Given an arbitrary positive integer x, what is the x^{th} digit in the decimal expansion of π ?
(P7) Given two arbitrary Pascal functions, do they compute the same function?
(P8) Given an arbitrary positive integer n and a polynomial P(x), what is the n^{th} derivative of P(x)?

We will say that a concrete question is an **instance** of the general problem it begets. It follows that any two concrete questions of the same kind are instances of the same general problem.

It should be fairly clear by now that what we need to formalize is the notion of a general problem, *not* that of a concrete question. For we set out to formalize the concept of a problem only as an intermediate step in our analysis of mechanical solvability; our ultimate goal is to find out which problems are mechanically solvable and which are not. And it is clearly of general problems that we want to find that out. We ask "is there a mechanical way to find the gcd of two (arbitrary) integers x and y?", we do not ask "is there a mechanical way to find the gcd of 12 and 30?". Even if we do ask the latter question what we really mean is "is there a mechanical way to find the gcd of two arbitrary integers x and y *that I can use to find the gcd of 12 and 30*?". Having thus made clear that the problems to which we can meaningfully attribute or deny mechanical solvability are general problems, we proceed to formalize these problems.

Our approach will borrow from the definition of functions. What is it that we need to know about a certain function $f : A \longrightarrow B$ to say that we know f completely and unambiguously? We need to know the value in the range B that f assigns to each object in the domain A. Now if f is finite we can do this by listing a finite number of equations which will provide the required information. For example, if the domain A is $\{a, b, c\}$ and $B = \{5, 9, 28\}$, we can give a perfectly complete description of a function f by writing down the equations

$$(I)\ f(a) = 5,\ f(b) = 9,\ f(c) = 28.$$

This is exactly the idea captured by the definition of a function as a set of

ordered pairs: the set $\{< a, 5 >, < b, 9 >, < c, 28 >\}$ gives us a thorough characterization of f by encapsulating all the information conveyed by equations *(I)*[3]. That is why we define a function $f : A \longrightarrow B$ as a subset of the Cartesian product $A \times B$ (with the usual uniqueness restriction).

In a similar way, what is it that we need to know about a general problem to say that we know it completely and unambiguously? We need to know (i) the input object(s) and (ii) the answer object *for each instance of the problem*, i.e for each concrete question of its type. A complete extensional description of problem *P1*, for example, would be an infinite list of the form

$$\begin{gathered} f(0) = 0 \\ f(1) = 1 \\ f(2) = 1 \\ f(5) = 1 \\ f(6) = 0 \\ f(7) = 1 \\ f(8) = 0 \\ \vdots \end{gathered}$$

where for every prime integer i the above list has an entry $f(i) = 1$, while for every non-prime integer j there is an entry $f(j) = 0$. Taking the set-theoretic approach, we can represent this general problem as the infinite set $\{< n, a > | \; n \in N, a \in \{0, 1\}\}$ where for each pair $< n, a >$, $a = 1$ iff n is prime. Likewise, we can represent problem $P8$ as the set $\{<< n, P(x) >, P^n(x) >\}$, where n is a positive integer, $P(x)$ is a polynomial of one real variable x, and $P^n(x)$ is the n^{th} derivative of $P(x)$. We can characterize problem # 3 with the set $\{<< n, m >, a > | \; n, m \in N \;\text{ and }\; a = gcd(n, m)\}$. Problem # 4 can be represented by the set $\{< \phi, a > | \; \phi$ is a wff of the propositional calculus, $a \in \{0, 1\}$, and $a = 1$ iff ϕ is satisfiable$\}$; and so on ...

In general, given any general problem P whose concrete questions take

[3]Of course if we limited our attention to finite sets we would not *need* to define functions as sets of ordered pairs; we could settle for the (arguably more intuitive) equation–list option, or something along those lines (like a table of values, for instance). However, the deciding cases are functions from and/or to infinite sets. Now with *some* such functions we can (and do) still go with the equational approach: when we describe the successor function as $f(n) = n + 1$, for example, we are giving an equational *schema* for the generation of an infinite number of equations of the form $f(1) = 2$, $f(2) = 3$, etc. However, for many infinite functions an equation schema (a "closed formula") of this sort is very difficult to find, and for even more infinite functions it can be proved that no such schemas exist. By contrast, sets are much more fundamental, ground-floor objects. They always "exist" (or at least held to exist) in a timeless, platonic fashion, regardless of whether they can be finitely described, or even of whether we ever come to actually "notice" them— which is why the present set-theoretic definition of function was adopted.

their inputs from k input domains $I_1, I_2, ..., I_k$ and have their answers in an answer domain A, we can represent P as the set $\{<< i_1, ..., i_k >, a > |\ i_j \in I_j,$ $1 \leq j \leq k,\ a \in A,$ and a is the answer to the concrete question determined by $i_1, i_2, \ \ldots, i_k\}$. But this is nothing but a binary relation whose domain is the Cartesian product of the input domains $I_1 \times \cdots \times I_k$ and whose range is A, the answer domain. Moreover, since any k given inputs determine a *unique* answer object (to each concrete question there is exactly one answer), the relation at hand is really *a function from* $I_1 \times \cdots \times I_k$ *to* A. This makes good sense because it enables us to think of a general problem as a mapping that assigns a unique answer in the answer domain A to each concrete object i (or k-tuple of input objects $< i_1, ..., i_k >$) in the input domain(s). We thus arrive at the following definition:

DEFINITION 2.1.1. *A (general) problem is a function from one or more input domains* $I_1, I_2, ..., I_k$ *to an answer domain* A.

Note that if we view a concrete question as an ordered pair $<< i_1, ..., i_k >, a >$, where $< i_1, ..., i_k >$ is a k-tuple of some mathematical objects and a is the answer, then our definition entails that a general problem is nothing but *the collection of all its instances* — which is intuitively appealing.

Our definition essentially conflates the notions of function and problem, at least from an extensional standpoint; it turns every problem into a function and every function into a problem. Intensionally, solving a problem now becomes computing a function and the computation of a function becomes the solution to a problem. This identification has traditionally been the distinct approach of Computer Science, where solving problems and computing functions amounts to the same thing. The reader is invited to evaluate this analysis. Does the definition make sense? Does it seem to be *accurate*? According to the distinctions we introduced in the previous chapter this is a mathematical entity definition, as it purports to tell us *what* a problem is. We argued that the accuracy of such definitions cannot be easily appraised and that only a combination of intuition and pragmatic considerations can, in due course, help us reach an informed conclusion. Nevertheless, we saw that a "minimal" requirement is that for each (clear) instance i of the definiendum there is an instance i' of the definiens which we can "think of" as i; and that for each instance i' of the definiens there is an instance i of the definiendum which we can "think of" as i'. Does our definition satisfy this requirement? Yes, it does. In the first direction, every general problem can be seen as a function from one or more input domain(s) to an answer domain. The rationale for this has already been discussed in the course of the analysis we gave in the last few pages. Now in the other direction, can every function be thought of as a general problem? Yes it can, because every func-

tion $f : A_1 \times \cdots \times A_k \longrightarrow B$, for arbitrary sets $A_1, ..., A_k, B$ $(k > 0)$, gives rise to at least one general problem: *the computation of its own values*, i.e the problem: "Given arbitrary $a_1 \in A_1, ...a_k \in A_k$, what is $f(a_1, ..., a_k)$?". It is due to this intrinsic "problem-hood" of functions that our definition does not become over-inclusive[4].

Thus we see that the identification of problems with functions makes a good deal of sense, at least at this first stage. In the following sections we will modify the definition to make it more suitable for our purpose, which is the development of computability theory. Such modification is not illegitimate, for we saw in section 1.3 that, depending on the context, one and the same class of entities (in our case general problems) can be equally well defined by two or even more definitions. The basic idea, however, will remain the same: general problems will be viewed as functions. The only change we will make will be to *narrow things down* to functions from the natural numbers to the natural numbers, i.e to the so-called **number-theoretic functions**. This focalization will prove to be incredibly beneficial. At present our definition might be sensible but it is extremely general: *any* old function from *any* domain(s) to *any* domain counts as a problem. Later, by defining problems as number-theoretic functions we will eliminate the need to consider arbitrary mathematical domains; we will instead restrict our attention to one single domain containing very simple and very familiar objects, the natural numbers. To see how and why this can be done, we must first say a few things about symbols and representation schemes.

2.2 Representation schemes

Everybody knows that we use **symbols** to denote mathematical objects, just as we use names to refer to people. The Romans used the symbol 'II', for example, to denote the number two. Nowdays we use the symbol '2' to denote the same object. Therefore, one should not confuse a symbol with what the symbol stands for—just as one should not confuse a name like 'George Bush' with what the name stands for, i.e with the actual *person* George Bush. 'II' and '2' are two obviously different symbols; but they denote the same thing, namely the abstract logical entity which is the number two. What *is*, then, the actual number two? No one knows and the chances

[4] Although it will inevitably be over-inclusive in the "benign" sense explained in section 1.3.

are no one ever will[5]. But the point is that we do not have to know what numbers are to reason about them, to add them, divide them, and use them to fill out our tax forms and build nuclear weapons. These pragmatic considerations imply that it is not irrational to assume, as we customarily do, that there *is* something that answers to the name '2', despite the fact that we will probably never know what that something is. Nobody really knows who George Bush is, either (indeed, nobody really knows —unless they have reached Nirvana— what his/her own name stands for). But that does not mean that it is irrational to assume that there is some*one* or some*thing* that does answer to the name "George Bush", as it is obviously useful and widely common to do so.

Two remarks about symbols are in order here before we continue.
(1) Sometimes we will want to distinguish between what a symbol s denotes on the one hand and s itself (as a syntactic object) on the other. Whenever we want to refer to the latter we will enclose s within single quotes (we have already been following this convention). Absence of quotes will thus mean that we are talking about the denotation of s. Hence, a statement such as " '10' has two digits" is meaningful (and true), whereas " 10 has two digits" is senseless because numbers, as abstract entities, do not have digits.
(2) Also, by 'symbol' we will not necessarily mean a single character out of a certain alphabet, but rather an arbitrary expression, sign, or mark having a finite size— including pictures, strings of characters, etc. For example, we call the formula '$(p \wedge q) \Longrightarrow r$' a symbol. By the same token, we would say that figure 2.3 depicts two different symbols (both of which denote the same object incidentally).

We usually give names to mathematical objects in a much more systematic and uniform way than we do to humans. A domain D of mathematical objects usually has at least one **representation scheme** R_D which allows

[5]Sure, we can give a logically satisfactory definition of 'two' in set theory as the object $\{\emptyset, \{\emptyset\}\}$, for example. Indeed, most mathematical objects can be made to boil down to what one calls "pure sets" is the set-theoretic universe. But that does not really cut any ontological ice; it merely pushes the bulge under the carpet only to have it re-appear at another point. For nobody really knows what a set is, either. Moreover, it has already been pointed out that within set theory there is usually more than one correct way to define mathematical objects like numbers. One could define the number two, for instance, in Von Neumman's way or in Russell's way. Who is to decide which one is the "real" number two? What is worse, a non-mathematician would probably reject both definitions as counter-intuitive. Again, the upshot is that in some cases the emphasis is not on *ontological* but rather on *functional* concerns: what really matters is not *what* mathematical objects are (something which we cannot reasonably hope to know, perhaps because there is nothing factual there to know), but rather *how* they behave (what laws they obey, etc.).

us to talk about objects in D. We will view a representation scheme R_D as a pair (S_D, M_D) comprising
(i) a set S_D of symbols, which will denote elements of D, and
(ii) a mapping rule M_D which makes clear which symbols in S_D stand for which objects in D.
One could think of M_D as an one-one function from S_D to D, and occasionally we will write $M_D(s)$ (for some symbol $s \in S_D$) to mean the object in D denoted by s; but for the most part we will view M_D informally as a "rule", or better yet as an implicit understanding of the referrents of S_D's symbols. To take an example, consider the domain of the natural numbers. One representation scheme for this domain is $R_N = (S_N, M_N)$, where S_N = { '0', '1', '2', ...,'738', ...}, i.e S_N consists of the usual Arabic decimal numerals, and M_N is the familiar mapping rule that associates zero to '0', one to '1', etc.[6] Of course one and the same domain might have more than one representation scheme. Another representation scheme for N, for instance, is $R'_N = (S'_N, M'_N)$, where S'_N = { '0', 'I', 'II', 'III', 'IV', 'V',... } contains the Roman numerals augmented with the symbol '0', and M'_N is the usual rule that maps 'I' to one, etc. As a last example, consider the domain T of all finite binary trees whose nodes contain, say individual (upper-case) letters of the English alphabet. One representation scheme $R_T = (S_T, M_T)$ for that domain is the customary graphical node-arrow representation of binary trees. S_T comprises all (two-dimensional) symbols like the one shown on the left side of fig. 2.3. A second scheme $R'_T = (S'_T, M'_T)$ uses lists to represent binary trees. Here the symbols of S'_T are one-dimensional: they are strings of characters built from the alphabet { '(', ')', 'A', 'B', ..., 'Z' }. Clearly, representation schemes are arbitrary human creations, adopted and upheld by social convention. Different cultures have used different schemes to represent the same domain of mathematical entities.

It should be clear that any mathematical domain that has ever been discussed by humans has at least one representation scheme. For, in the absence of such a scheme we cannot even talk about the domain's objects (probably not even think about them), just as in the absence of a name or some other finite verbal description we cannot talk about a person. Whenever we utter something about a mathematical object, or write something about it, or have a thought about it, we are always using a certain finite symbol to represent the object, and that finite symbol is usually one of a countable infinity of similar symbols that share a common syntactic structure.

[6] In the sequel we will consistently use S_N to mean the set of symbols '0', '1', '2', ...

This last remark brings us to the next point, which concerns the cardinality of a symbol set S_D. In particular, from our stipulation that all symbols in S_D must be finite it follows that S_D has to be countable[7]. In other words, S_D is always either finite or countably infinite, i.e equinumerous to N. The former case is rather rare. Finite symbol sets are used only when the underlying domain D is an arbitrarily gerrymandered finite collection of divers mathematical objects. We might be dealing, for example, with a function whose range consists solely of π and the square root of 0.5. Then the symbol set for this domain might be { 'π','$\sqrt{\frac{1}{2}}$'}. Finite symbol sets such as this usually comprise arbitrary symbols that do not share a common syntax pattern; such symbol sets do not have *syntactic cohesion*. In the great majority of cases, however, we deal with countably infinite symbol sets like S_N = {'0', '1', ...}. Symbol sets of this kind are syntactically homogeneous, i.e their symbols are similar-looking, a fact that reflects the semantic cohesion of the underlying domain. The elements of such symbol sets are orderly specified in accordance with precise *formation rules*. Usually such rules are recursive and can be described by simple context-free grammars. In most cases we start with a small number of *elementary* ("atomic","primitive") symbols and then we recursively build on those to generate more complex ("molecular", "compound") symbols. For example, the elementary symbols of S_N are the digits '0' through '9'. More complex symbols (numerals with two or more digits) are constructed by concatenating finitely many elementary symbols, with the rule that no compound symbol should begin with the digit '0'. The set of rational numbers Q can be represented with a scheme (S_Q, M_Q) where each symbol in S_Q is obtained by inserting a slanted line / in between two symbols s and s' of S_N (with the stipulation that s' should not be the digit '0'), and M_Q is the customary fraction interpretation of these expressions. In this case the atomic ingredients of S_Q are the digits '0', ..., '9' along with the symbol '/'. In S_T (where T is the domain of binary trees we mentioned earlier) the primitive symbols are 26 circular nodes, each containing a different (upper-case) English letter. Then we have a (recursive) rule which says that given any symbol $s \in S_T$ (primitive or not), we can obtain a new symbol s' by attaching a left node (and an associated edge), or a right node, or both, to any one of s's leaves (a leaf is a node with no children). Thus we

[7]This should be obvious, but if it is not, here is an informal *reductio ad absurdum* for it: if each symbol s in S_D is finite, there must be a finite English string $ST(s)$ (i.e a finite sequence of English sentences) giving a thorough, exhaustive description of s. For different symbols s and s', $ST(s)$ and $ST(s')$ would also be different. Therefore, if S_D contained uncountably many finite symbols s, there would be uncountably many English strings $ST(s)$– which is impossible.

get a countable infinity of finite symbols (in this case the symbols are diagrams). In S'_T the compund symbols are built from the elementary tokens ')', '(', 'A', 'B', ..., 'Z'. And so on...

Now the fact that representation schemes must always be based on countable symbol sets does not limit their utility. For, any mathematical domain D is either countable or it is not. If it is countable then there is a bijection between the symbols of S_D and the objects of D, and hence R_D is perfectly expressive. That is the case, for example, with N and $R_N = (S_N, M_N)$. But even when D is *un*countable we get along just fine with countable representation schemes. Take the real numbers, for example. Most of them are irrational, but in practice we deal almost exclusively with rationals— which are countably many and can therefore be represented perfectly well with a countable representation scheme (using, for instance, finite strings of digits with embedded periods). The reason for this is that irrational numbers are infinite objects which we cannot write down (or perhaps even conceive) in full detail. Even when we need to talk about a "natural" irrational number, such as the logarithm base e, we usually resort to one of two avenues: if the value of the irrational number is needed in a computation, we use an approximation of it, which is obviously bound to have a finite and hence rational magnitude; or if the value is not needed, we simply tag an arbitrary finite symbol on the number, i.e we give it a name like π or e or $\sqrt{2}$, etc. In either case we use a finite symbol to denote the number. As another example, the same type of considerations apply to the (uncountable) domain F of all polynomial functions of one real variable x and its usual (countable) representation scheme which uses symbols like $3.78x^2 - 4x + 102.99$.

We summarize the discussion of the last few paragraphs as follows:

EMPIRICAL PROPOSITION 2.2.1. *Every domain D of mathematical objects has a representation scheme $R_D = (S_D, M_D)$. Furthermore, S_D comprises countably many distinct finite symbols.*

The object/symbol distinction naturally carries over to questions and problems. Let P be any general problem. According to our earlier definition, P is a function from one or more input domains $I_1, I_2, ..., I_k$ $(k > 0)$ to an "answer domain" A. From proposition 2.2.1, both the input domains and the answer domain have certain representation schemes $R_1 = (S_1, M_1), \ldots, R_k = (S_k, M_k), R_A = (S_A, M_A)$. In practice, whenever we tackle an instance of P (a concrete question) what we are given as input are not actual objects from the input domains (for, as was already said, noone really has a handle on such objects), but rather *symbols* $s_1, ..., s_k$ (from $S_1, \ldots, S_k$ respectively) *that denote objects in* $I_1, \ldots, I_k$. Similarly, what we try to produce during the solution process is a symbol $s \in S_A$ denoting

a certain object in the answer domain. Hence, although we may sensibly view a concrete question abstractly as an ordered pair whose first element is a k-tuple of mathematical objects and whose second element is another abstract object, *in practice we always deal with a symbolic representation of the question*, which can be viewed as a pair whose first element is a k-tuple of symbols and whose second element is also a symbol. Generalizing from concrete questions to general problems yields the following definition:

DEFINITION 2.2.1. *Let $P : I_1 \times \cdots \times I_k \longrightarrow A$ be a general problem and let $R_1 = (S_1, M_1), \ldots, R_k = (S_k, M_k)$ and $R_A = (S_A, M_A)$ be representation schemes for the k input domains and the output domain, respectively. Then a* **symbolic representation** *of P is a function $R_P : S_1 \times \cdots \times S_k \longrightarrow S_A$ such that for all symbols $s_i \in S_i$ $(1 \leq i \leq k)$ and $s_a \in A$,*

$$R_P(s_1, \ldots, s_k) = s_a \Longleftrightarrow P(M_1(s_1), \ldots, M_k(s_k)) = M_A(s_a).$$

As should be expected, this definiton implies that a general problem can have several symbolic representations. The general problem "What is the gcd of two arbitrary non-negative integers?", for example, might be symbolically represented as a function from $S_N \times S_N \longrightarrow S_N$, or as a function from $S'_N \times S'_N \longrightarrow S'_N$, or as a function from $S_N \times S'_N \longrightarrow S_N$, etc. (recall that S_N and S'_N are the symbol sets {'0', '1', '2', ...} and { '0', 'I', 'II', ... }, respectively).

2.3 Arithmetical representation schemes. Arithmetical representations of problems

Let D be a mathematical domain and let $R_D = (S_D, M_D)$ be a representation scheme for it. In what follows we will assume that S_D is (countably) infinite. From what was said in the previous section, this is a pretty realistic assumption. At any rate, all we are about to say will be easily seen to apply to finite symbol sets as well. Assuming, then, that S_D is countably infinite, there is a bijection between it and N. Suppose we had figured out a method, an algorithm, call it AL_D[8], to construct such a bijection, i.e a method for assigning a unique number to each symbol in S_D and a unique symbol to each number. Then instead of using the symbols of S_D to talk about the objects of D, we could simply use numbers. If s, for example, is some particular symbol in S_D, and if 35 is the unique number code that our method AL_D assigns to s, then in any stretch of discourse involving D we

[8]Of course we do not yet *formally* know what an algorithm is, but this is not a problem. An informal understanding of the notion is perfectly sufficient for the purposes of this section.

can just say '35' and understand by it the object in D that is denoted by s. In other words '35' is now an alias, a second name for s and can be used *in place of* s to denote the same object in D that s denotes.

Of course to be able to freely use numbers in place of the symbols of S_D our algorithm AL_D must provide us with:
(1) a recipe-like **encoding** procedure which will compute the number code of *any* given symbol $s \in S_D$ in a finite amount of time, and
(2) a similar **decoding** procedure which will enable us to retrieve the symbol $s \in S_D$ encoded by *any* given number n. If such a method AL_D is available, then we can freely use numbers to encode symbols in S_D and hence, transitively, objects in D.

As an example, consider the domain F of all polynomial functions of one real variable x. One representation scheme for this domain is $R_F = (S_F, M_F)$ where S_F consists of all symbols (expressions) such as[9]

$$\textbf{(I)}\ 3.5x^4 + 2.897x^2 - 39x + 76.2, \text{ or}$$
$$\textbf{(II)}\ 1004.9x^{197} - 22x + 3.14, \text{ etc.},$$

and M_F is the usual understanding which associates such expressions to the infinite objects (polynomial functions) they denote.

Now suppose we had a recipe-like method AL_F for assigning a unique number n to each expression in S_F, so that every number encoded some symbol in S_F. Then we can bypass R_F and use numbers to denote polynomials. If, for instance, 39 and 11^{28} were the (unique) numbers assigned to expressions **(I)** and **(II)** respectively, then we can just use 39 and 11^{28} to denote the same functions that **(I)** and **(II)** denote. Essentially AL_F would give rise to a second representation scheme $AR_F = (S_N, AL_F)$, where S_N = { '0', '1', '2', ... }, which would allow us to use numbers (srtictly speaking, numerals) to refer to objects in F (roughly as we use numbers to refer to TV channels; 13 for NBC, 6 for CBS, etc.). For any mathematical domain D, let us say that an **arithmetical representation** of D is a representation scheme $AR_D = (S_N, AL_D)$. The following proposition is crucial to our theme:

EMPIRICAL PROPOSITION 2.3.1. *Every mathematical domain D has an arithmetical representation scheme.*

This is a result of the following :

EMPIRICAL PROPOSITION 2.3.2. *For every symbol set S_D there is an algorithm for mapping the symbols of S_D onto the natural numbers in an one-to-one fashion. The algorithm is such that given any symbol $s \in S$ we can find its (unique) code number n in a finite amount of time; and given*

[9]The definition of S_F could be easily made precise with a BNF-like grammar.

any number n, we can also find the (unique) symbol s encoded by n in a finite amount of time.

It is easy to see that 2.3.1 follows from 2.3.2. For, let D be an arbitrary domain. By proposition 2.2.1, D has a representation scheme $R_D = (S_D, M_D)$. Then 2.3.2 implies that there is an algorithm AL_D which maps the symbols of S_D onto the natural numbers. Therefore, $AR_D = (S_N, AL_D)$ is an arithmetical representation scheme for D.

Proposition 2.3.2 plays a central role in computability theory. The main idea behind it is an ingenious technique for encoding arbitrary finite expressions by numbers, known as **Godel numbering**, in honor of its originator K. Godel. The method is based on the fact that finite sequences of numbers can be uniquely encoded by single numbers. We discuss this topic in detail in section 4.1; a (very important) working example is given later in section 4.2, where we use Godel numbering to **arithmetize** Turing programs, i.e encode them by numbers. Now proposition 2.3.2 is empirical and, as such, it can be affirmed only through experience and observation, *not* by means of a deductive argument[10]. However, once one has become familiar with Godel numbering and has witnessed several concrete applications of it to various sets of symbols, one will not hesitate to make the inductive leap and conclude that the method can be applied to *any* set of symbols in *any* representation scheme. The truth of 2.3.2 will then be seen to be beyond doubt. The reader who has had no experience whatsoever with encodings should therefore suspend his/her judgement at least until section 4.2.

Returning to our main theme, it should be emphasized that an arithmetical representation scheme $AR_D = (S_N, AL_D)$ is always an *indirect* scheme for representing objects in D, i.e it is always derived from some other intermediate representation scheme for D.[11] In other words, to construct an arithmetical representation scheme for D we must first have *another* representation scheme $R_D = (S_D, M_D)$ and then discover an algorithm AL_D that will 1-1 map the symbols of S_D onto N (the existence of such an algorithm is guaranteed by proposition 2.3.2). But once this is done, we can "ignore" R_D and think of AL_D as a rule that maps numerals in S_N directly to objects in D; hence we will write $AL_D(\text{`5'})$, for example, to denote a certain object in D. The example we will give in chapter 4 will help clarify this further.

The notion of arithmetical representation schemes is naturally extended from domains to general problems, since the latter are simply functions on

[10] At least as formulated here.

[11] The obvious exception to this is the domain N, since the natural representation scheme for this domain *is* arithmetical.

domains:

DEFINITION 2.3.1. *Let $P : I_1 \times \cdots \times I_k \longrightarrow A$ be a general problem and let $AR_1 = (S_N, AL_1), \cdots, AR_k = (S_N, AL_k)$, $AR_A = (S_N, AL_A)$ be representation schemes for the k input domains and the answer domain, respectively. Then an* **arithmetical representation** *of P is a function $AR_P : S_N^k \longrightarrow S_N$ such that for all numerals $s_1, \ldots, s_k, t$ in S_N, we have*

$$AR_P(s_1, \ldots, s_k) = t \Longleftrightarrow P(AL_1(s_1), \ldots, AL_k(s_k)) = AL_A(t).$$

Since one domain can have several arithmetical representation schemes, a general problem can also have several arithmetical representations. Eventually we will see that these different representations are isomorphic in an important sense. Finally, proposition 2.3.1 implies the following:

EMPIRICAL PROPOSITION 2.3.3. *Every general problem has at least one arithmetical representation.*

2.4 Definition of general problems as nt functions. Partial functions and new definition.

At this point it will be expedient to become a little lax about the object/symbol distinction and view the arithmetical representation AR_P of a problem P as a function from N^k to N (i.e from numbers to numbers) rather than from S_N^k to S_N (from symbols to symbols). We will thus speak directly of nt functions as arithmetical representations of problems. This should not cause any confusion because for every function $f : N^k \longrightarrow N$ there is *exactly one* corresponding function from S_N^k to S_N, and vice versa.

With this understanding, proposition 2.3.3 of the preceding section tells us that every general problem can be "arithmetically represented" by an nt function $f : N^k \longrightarrow N$. The converse is also true, i.e every function $f : N^k \longrightarrow N$ represents a general problem, namely "given arbitrary integers $n_1, \ldots, n_k$, what is $f(n_1, \ldots, n_k)$?". Hence the class of all nt functions contains all and nothing but the general problems (or, rather, arithmetical representations thereof). This is the motivation for conflating the two notions in the following way:

DEFINITION 2.4.1. *A general problem is a function $f : N^k \longrightarrow N$.*

By this definition, to answer a concrete question is to compute a particular value $f(n_1, \ldots, n_k)$ of some particular function f. The usefulness of this formalization derives from the tremendous extent to which it delimits the definiendum. In our explorations of mechanical solvability we no longer have to consider problems in general, as mappings from arbitrary domains to arbitrary domains. Our task is immensely narrowed: all we now have to

do is focus on functions from the natural numbers to the natural numbers—very straightforward, very familiar objects.

Because definition 2.4.1 plays such an important role in our development, let us briefly recapitulate the rationale behind it. Since this is a mathematical entity definition, we cannot expect either strict or "fuzzy" extensional identity between the definiendum and the definiens. All we can expect from it in the way of accuracy is that it be a good ontological "proposal". And we saw that a proposal is accurate in that respect only if it makes sense to conflate definiendum and definiens, i.e to "think of" instances of the former as instances of the latter and vice versa. Now to show that this is *not* the case for our definition, one would have to demonstrate either under– or over–inclusiveness. To do the first, one must adduce a general problem which we cannot "think of" as an nt function. But proposition 2.3.3 assures us that this is impossible. To prove over-inclusiveness, one must pinpoint an nt function which does not represent any general problem. But this is not possible either, because any given function $f : N^k \longrightarrow N$ represents a general problem in its own right: the computation of its values. Hence the accuracy. It is noteworthy, however, that this argument depends, as does the definition itself, on proposition 2.3.3. If that proposition is false, i.e if there is a general problem which does not have an arithmetical representation, then our definition is under-inclusive and therefore inaccurate. Now proposition 2.3.3. itself depends on proposition 2.3.1; and proposition 2.3.1, in turn, holds only if proposition 2.3.2 holds. Thus our entire analysis reduces to and depends upon the truth of the empirical assertion that all symbol sets can be arithmetized. As we have already remarked, once one becomes familiar with Godel numbering and other similar encoding/decoding techniques, the truth of this assertion becomes evident—although it can never be proven mathematically.

Unfortunately (and this might come as a surprise at this point), although definition 2.4.1 has the right idea, it cannot serve as the basis of a theory of computation yet; it will have to be *slightly* amended first. The reason for this is technical and will be explained in full detail in Appendix A. It has to do with **total** and **partial** functions. Observe that we have tacitly defined an nt function f to be a *total* function from N^k $(k > 0)$ to N, meaning that for *every* k-tuple of integers $n_1, \ldots, n_k$, f is defined and has some specific number $f(n_1, \ldots, n_k)$ as its value. However, it turns out that if we restrict our attention to total nt functions it becomes impossible to formalize mechanical solvability. It is provably impossible to find an *accurate* definition of "mechanically computable nt function" if we take nt functions to be total. A technique known as diagonalization will always produce a

counter-example that will show *any* proposed formalization to be under-inclusive. (Again, a full explanation of this is given in Appendix A). The usual way around this (due to Kleene) is to extend our notion of a problem by admitting *partial* nt functions into the picture: a partial nt function $f : N^k \longrightarrow N$ is a function that might be *undefined* for *some* (zero or more) of its arguments, meaning that for some k-tuples $n_1, \ldots, n_k$, f might have *no value*, indicated as $f(n_1, \ldots, n_k) \uparrow$ and read as "f is undefined for $n_1, \ldots, n_k$". (We indicate that f *is* defined for $n_1, \ldots, n_k$ by writing $f(n_1, \ldots, n_k) \downarrow$). An example is the function

$$f(x) = \begin{cases} x+1 & \text{if } x \text{ is even} \\ \uparrow & \text{if } x \text{ is odd} \end{cases}, \text{ which is undefined for all odd numbers, or}$$

$$g(x,y) = \begin{cases} x-y & \text{if } x \geq y \\ \uparrow & \text{otherwise.} \end{cases} \quad \text{(the "partial subtraction" function).}$$

A rather special partial function that we will frequently have occasion to consider is the **nowhere-defined function** $h(x) = \uparrow$, which is undefined for all numbers. From now on we will use the abbreviation "pnt function" as a shorthand for "partial number-theoretic function". Note that a total function *is* a partial function which happens to be everywhere defined. Therefore, when we speak of a pnt function f, it should not be automatically inferred that f is total, *unless* we explicitly say so. We now extend our first definition as follows:

DEFINITION 2.4.2. *A general problem is a* partial *function* $f : N^k \longrightarrow N$.

Note that this second definition does not *contravene* our first definition in any way, because, from what we said above, total nt functions *are* partial functions. In other words, the extension of the first definiens is a sub-class of the extension of the second definiens. Hence, since we saw that the first definition was not under-inclusive, the second one cannot be under-inclusive either. The only new risk is over-inclusiveness, since the definiens has now been extended by the inclusion of additional objects, namely the non-total functions. But one can easily see that this does not create a problem, for just as every total function was seen to determine at least one general problem, the computation of its values, every partial function can likewise be seen to determine a similar general problem: the computation of its values $f(n_1, \ldots, n_k)$ *for those arguments* $n_1, \ldots, n_k$ *for which* $f(n_1, \ldots, n_k) \downarrow$. It follows that if the first definition is accurate, then our amendment is harmless. Moreover, it is dictated by pragmatic factors; it serves to overcome the diagonalization obstacle, which will be seen to be insurmountable if we choose to treat total functions only.

Chapter 3

Formalizing algorithms

In the previous chapter we hypostasized general problems as pnt functions. By virtue of that formalization, solving a problem has become tantamount to computing a pnt function. As a result, our original task of discovering which problems are mechanically solvable has been reduced to discovering which pnt functions are mechanically computable. Therefore, what is now needed is a mathematical predicate definition of the form "A pnt function is mechanically computable iff ...", where the dots are to be filled with an evidently sharp concept. From what was said in the first chapter, such a definition will be accurate iff the definiens comprises all and only those pnt functions which are intuitively deemed to be computable.

In this chapter we will offer not one but three such definitions:
(1)A pnt function is mechanically computable iff it is Pascal-computable.
(2)A pnt function is mechanically computable iff it is recursive.
(3)A pnt function is mechanically computable iff it is Turing-computable.
The formalization we will adopt and use for the rest of the essay will be the third one, that of Turing machines. All three definitions, however, turn out to be extensionally equivalent, meaning that their definientia are co–extensive. We end the chapter with an enunciation of Church's thesis and a discussion of its practical ramifications.

3.1 First Formalization: Pascal-computability

Let $f : N^k \longrightarrow N$ be a pnt function of $k > 0$ arguments. We say that f is **Pascal-computable** iff there is a Pascal subroutine[1] P of INTEGER type,

[1]By 'subroutine' I will exclusively mean a Pascal FUNCTION, *not* a PROCEDURE.

having exactly k INTEGER parameters, such that for all numbers $x_1, \ldots, x_k$,
(1) If $f(x_1, \ldots, x_k) \downarrow$, then when P is invoked with actual input arguments $x_1, \ldots, x_k$, its computation eventually **halts** (**"converges"**), and the last assignment statement executed is of the form "P := e;" where e is a Pascal expression of INTEGER type whose value is $f(x_1, \ldots, x_k)$.
(2) If $f(x_1, \ldots, x_k) \uparrow$, then when P is started with actual arguments $x_1, \ldots, x_k$ it gets into an infinite loop and thus fails to halt. In that case we say that P **diverges** on input number(s) $x_1, \ldots, x_k$.
If such a Pascal function P exists, we say that P **computes** f.

To put it another way, f is Pascal-computable iff it is possible to write a Pascal subroutine which will compute the value $f(x_1, \ldots, x_k)$ for all those arguments $x_1, \ldots, x_k$ for which f is defined, but will diverge for all input numbers for which f is undefined. As usual, by 'numbers' we mean natural numbers (0,1,2,...). We do not care how P behaves when one or more of its actual arguments are negative integers.

To take an example, consider the function

$$f(x) = \begin{cases} 3x^2 + 5 & \text{if } x > 10 \\ \uparrow & \text{if } x \leq 10 \end{cases}$$

The following subroutine proves that f is Pascal-computable:

```
FUNCTION DoIt(x : INTEGER) : INTEGER;
BEGIN
    IF (x > 10) THEN
        DoIt := (3 * (x * x)) + 5
    ELSE
        WHILE (x = x) DO
                ;          (* empty statement *)
END;
```

It is easy to see that for any input argument x, DoIt gets into an infinite loop (diverges) iff $x \leq 10$, while it outputs the value $3x^2 + 5$ iff $x > 10$. Thus DoIt computes f, which, of course, makes the latter Pascal-computable. As another example, consider the nowhere-defined function $g(x) = \uparrow$. The following Pascal function computes g:

```
FUNCTION Nowhere(x : INTEGER) : INTEGER;
BEGIN
    WHILE (1 > 0) DO
            ;              (* iterate indefinitely *)
END;
```

Clearly, Nowhere fails to halt no matter what input argument it is given.

As a third example, consider the function

$$h(x) = \begin{cases} \uparrow & \text{if } x \text{ is even} \\ x! & \text{if } x \text{ is odd} \end{cases}$$

For instance, $h(3) = 6, h(5) = 120, h(20) \uparrow$, etc. The following computes h :

```
FUNCTION Odd_Fact(x : INTEGER) : INTEGER;
VAR
product, i : INTEGER;
BEGIN
    IF (x MOD 2) = 0 THEN
       REPEAT                          (* If x is even, get into *)
            x := x;                      (* an infinite loop *)
       UNTIL (1 <> 0)
    ELSE BEGIN
       product := 1;
       FOR i := 1 TO x DO              (* Otherwise, compute x! *)
            product := product * i;
       Odd_Fact := product;
    END;
END;
```

Note that Odd_Fact also halts and outputs a certain value for *negative* odd input integers. But as we have already pointed out, that does not concern us. It could just as well diverge on such integers, or diverge on some of them and halt on others, or, indeed, behave in any manner whatsoever. All we care about is the behavior of the subroutine on *non-negative* integer arguments. As long as it (1) diverges for all and only those natural numbers x for which $h(x) \uparrow$, and (2) converges and outputs $x!$ for all and only those numbers for which $h(x) \downarrow$, we will say that Odd_Fact *computes* h regardless of what happens for negative arguments.

Next we will delineate a few subtleties which *must* be thoroughly understood by anyone who wishes to attain a non–superficial grasp of computability theory. The Pascal formalism is a good medium for conveying these subtleties because it is contemporary, high–level, and easy to understand. Nonetheless, all of the remarks we are about to make apply equally well to the two subsequent formalizations and, indeed, to all conventional models of computation. They are all a little peculiar, they have intricate philosophical edges, and they take us to the heart of the notion that we are studying.

1. **No constraints whatsoever are imposed on time and space**

requirements. In particular, we impose no bounds on the size of the input numbers, the size of the output number, the number and size of local variables, the length of identifiers, and the time length of the converging computations. All we require of these quantities is that they be finite. Other than that, they can be *arbitrarily large*. Furthermore, we are not at all concerned about overflows, underflows, hardware faults, power failures, CPU speed, or any other empirical factors. In other words, we assume that our Pascal subroutines are executed on (1) an idealized computer which can store and manipulate arbitrarily large numbers without any hardware risks, and by (2) an idealized computing agent ("computist") who is willing and able to wait an arbitrarily long time for the answer.

These assumptions might seem unrealistic at first, but they are in fact perfectly sensible.[2] For it is easily seen that *any* upper bound we might suggest for *any* of the above quantities will be wholly arbitrary and thus ultimately meaningless. It would be similar to number theorists postulating the existence of a greatest positive integer in the futile hope of blocking out inordinately large numbers— and it would thus be prey to similar objections. At any rate, constraints and bounds that seem reasonable today are likely to become ludicrous tomorrow. A hundred years ago the computation of the fist ten million digits of π was a formidable computational task, yet today it is almost trivial. By foresaking practical considerations we will able to disentangle our inquiry from any particular state-of-the-art of human technology and achieve *theoretical* rather than practical limitations on mechanical computability. Our results will thus be as valid twenty thousand years from now as they were twenty thousand years ago.

2. **One and the same fuction can be computed by seberal subroutines.** The factorial function $f(x) = x!$, for example, can be computed by a recursive subroutine or by an iterative one (see fig. 2.2). The function $gcd(x, y)$ can be computed by brute force, or by Euclid's algorithm, or in several other ways. Two subroutines that compute the same pnt function are intensionally different (they embody different methods, different algorithms) but extensionally identical, meaning that for same inputs they produce the same results.

   We can give a very simple argument to prove that *every* Pascal-

[2] And they are also typical of theoretical investigations—consider the idealized pendulum in mechanics.

computable function can be computed by more than one subroutine. In fact we can do much better than just "more than one" :

PADDING LEMMA. *If a pnt function f is Pascal-computable, then there are infinitely many subroutines that compute f.*

Proof. Since f is Pascal-computable, there must be a Pascal function P that computes it. Let a be the name of the first parameter of P and let P_1 be the subroutine obtained from P by inserting the irrelevant but innocuous statement "a := a;" immediately after the BEGIN line of the main body of P. Now P and P_1 are two distinct Pascal subroutines because they are syntactically different. But it is obvious that they are extensionally identical, meaning that they both compute the same nt function. Now let P_2 be the subroutine obtained from P_1 in the same way P_1 was obtained from P. Do the same to P_2 to get a new subroutine P_3, and then a P_4, and so on. These P_is, $\mathsf{i} > 0$, comprise a set of infinitely many distinct Pascal subroutines that compute the same function, namely f. ■

One can see why this result is called "the padding lemma" : we repeatedly "padd" the text of a given subroutine P with one or more superfluous statements to produce an indefinite number of different Pascal subroutines whose extensional behaviors are identical to that of P.

An interesting question that may be posed in this connection is: Can we write a subroutine that will take as input two arbitrary Pascal subroutines (stored, say, as text files) and tell us whether or not they compute the same nt function? We will soon (corollary 4.4.1) prove that this is impossible, i.e that no such subroutine exists!

3. **All finite pnt functions are Pascal-computable.** A finite pnt function is one with a finite domain, i.e one which is defined only for finitely many numbers. e.g

$$f(x) = \begin{cases} x+1 & \text{if } x \leq 10 \\ \uparrow & \text{if } x > 10 \end{cases} \quad \text{or } g(x) = \begin{cases} 2^x & \text{if } x \text{ is prime and } x < 10^{89} \\ \uparrow & \text{otherwise.} \end{cases}$$

It is very important to realize *why* such functions are Pascal-computable. Here is the argument : let f be a finite pnt function of one argument[3] and let $D_f = \{d_1, \ldots, d_m\}$ be its finite domain, for some $m \geq 0$.[4] Now

[3] Our reasoning can be easily modified to handle k-ary functions for $k > 1$.

[4] For $m = 0$ f is the nowhere-defined function.

consider the following Pascal subroutine:

```
FUNCTION FiniteDom(x : INTEGER) : INTEGER;
VAR
Defined_For_x : BOOLEAN;
BEGIN
Defined_For_x := False;
IF x = d1 THEN BEGIN Defined_For_x := True; FiniteDom := f(d1);
END;
IF x = d2 THEN BEGIN Defined_For_x := True; FiniteDom := f(d2);
END;
⋮
IF x = dm THEN BEGIN Defined_For_x := True; FiniteDom := f(dm) ; END;
IF NOT Defined_For_x THEN
        WHILE (True) DO
                ;
END;
```

Strictly speaking, of course, as we have written it here, FiniteDom is not an actual Pascal subroutine; it is more of a subroutine *schema*. This is because in each IF statement we have used the two expressions d_i and $f(d_i)$ $(i = 1, 2, \ldots, m)$ as "dummies". To get actual Pascal code we must replace these dummies with their appropriate numerical values. For a concrete example, if the function is

$$f(x) = \begin{cases} 100x + 35 & \text{if } x < 5 \\ \uparrow & \text{otherwise} \end{cases}$$

then the third[5] IF statement of FiniteDom would be

```
IF (x=2) THEN BEGIN Defined_For_x := True;FiniteDom := 235;END;
```

Bearing this in mind, it is clear that FiniteDom computes f. The routine halts and outputs the correct value for all input numbers for which f is defined and gets into an infinite loop for all the (infinitely many) numbers for which f is undefined. By the same token one can see that similar subroutines exist for all finite functions f. No matter how large the domain of f might be, as long as it is finite there will be some Pascal function P similar to FiniteDom that computes f. Of course, depending on the size of f's domain, P might consist of billions or even trillions of IF statements. But that does not bother us.

[5]There would be six IF statements altogether, the first five for the five numbers in f's domain (0,1,2,3,4) and the sixth one leading into the infinite loop.

According to our definition, all we demand in order to pronounce f Pascal-computable is the *existence* of a Pascal subroutine that computes f— and this is where the subtlety comes in. For, the existence of such subroutines is entirely timeless and platonic, independent of and detached from practical considerations and the physical world.

It might be worthwhile at this point to elaborate a litlle further on "existence". Perhaps an example will help to clarify the idea. Consider the set of strings E^*, where E is the set comprising the upper- and lower-case letters of the English alphabet along with the usual punctuation symbols (the comma, the period, etc.) and a "blank" character. E^* is the set of all finite strings that can be formed by concatenating together arbitrarily many symbols from E. Thus E^* contains both gibberish (e.g strings like "..?-mX" or "gN.,!-Na""T") and meaningful strings (such as "The dog chased the cat down the hall."). In fact E^* contains all legitimate English sentences, as well as all finite sequences of such sentences separated by periods. Tolstoy's entire "War and Peace", viewed as a finite (albeit very long) string of characters from E, is an element of E^*. Now for every person P on earth, including those that were, those that are, and those that will be, there is a (very long) string S(P) in E^* which describes the life of P in every little detail from the moment they were born all the way to their death. Even though I am presently alive, for example, and my future is to a certain extent uncertain, nevertheless *there exists* a string in E^* which details on a day-to-day basis exactly how my life will turn out, where, when and how I will die, etc. Because *no matter how* my life turns out, it will ultimately be describeable to an arbitrary level of detail by a particular finite sequence s of English sentences; and s is nothing but an element of E^*, a finite concatenation of symbols from E. As such s *exists* now as it existed a million years ago and as it will always exist. Likewise, one can also find in E^* the most famous novel of the 21^{st} century, the whole truth about JFK's assassination (and proof for it), a detailed chronicle of American history in the 26^th century A.D., etc. This is what is meant when it is said that such strings exist in a timeless, static fashion. Now the same is the case with Pascal programs, which can themselves be regarded as long strings of symbols from $E' = E \cup \{$'0','1', ..., '9'$\}$, and which consequently exist or fail to exist in an absolute sense. When this is understood, it is plain to see that for every finite function f there is a (perhaps *very* long) string in $(E')^*$ which *happens* to be a Pascal function that computes f.

4. **All constant functions are Pascal-computable.** A constant function is a total nt function $f : N^k \longrightarrow N$ such that for all numbers $x_1, \ldots, x_k$, $f(x_1, \ldots, x_k) = c$ for some fixed number c. Examples: $f(x) = 5$, $g(x_1, x_2) = 2^{873}$, $h(x_1, x_2, x_3) = 17$. It is easy to see that all such functions are Pascal-computable. The above function h, for example, can be computed by the following subroutine:

```
FUNCTION H(x,y : INTEGER) : INTEGER;
BEGIN
    H := 17;
END;
```

Likewise, g can be computed by

```
FUNCTION G(x,y,z : INTEGER) : INTEGER;
BEGIN
    G := 2 * 2 * ··· * 2;   (873 times)
END;
```

Of course G's value is huge and its actual computation would cause an overflow on any contemporary computer. But for reasons we have already discussed, that is immaterial— g is Pascal–computable *in principle*, if not in today's practice. That goes to show that *all* constant functions, regardless of their values, are Pascal-computable.

3.2 Second formalization: Kleene's recursive functions

Our first formalization, the identification of mechanical computability with Pascal-computability, seems to have a good deal of initial credibility; most people are likely to have a "gut feeling" that if a pnt function can be computed mechanically at all, then there must be some Pascal subroutine or other that computes it (the converse being obvious). The plausibility of our second formalization, however, will not be as apparent. Because Kleene's definition is not cast in terms of a **model of computation** such as Pascal programs or Turing machines, it has no salient intensional relevancy to the theme of executing instructions in the step-by-step manner that people instictively associate with algorithmic computation. We might say that his definition characterizes the class of mechanically computable functions *descriptively* rather than *operationally*. But since his definition proves to be extensionally equivalent to the operational ones, its descriptiveness is ultimately an advantage rather than a drawback. It makes us see the class in

question from a different perspective, and it provides us with a refined and orderly description of its internal structure.

Central to Kleene's approach is the idea of defining a pnt function by successively applying a small number of definitional schemata to one or more **initial functions**.[6] The initial functions are the following:

1. The successor function $s(x) = x + 1$,
2. The "zero" function $z(x) = 0$, and
3. For each $k > 0$, a group of k "projection" functions $p_k^1, p_k^2, \ldots, p_k^k$ from N^k to N, where for each $i = 1, \ldots, k$, $p_k^i(x_1, \ldots, x_k) = x_i$.

These initial functions serve as the starting material for the definition of more complex functions through the operations of (1) composition, (2) recursion, and (3) unbounded minimalization.

(1) **COMPOSITION.** If we have two partial functions f and g (assume for simplicity that they are both of one argument), then we can *combine* them to define a new function h by setting $h(x) = f(g(x))$. Thus the output of g becomes the input to f and then the output of f becomes the value of h. Of course if g is undefined for some x, or if f is undefined for $g(x)$, then h will be undefined for x as well. More specifically, h will be total iff g is total *and* f is defined for all numbers in g's range.

In the more general case of multiple variables, if we have $n > 0$ functions $g_1, \ldots, g_n$ of $k > 0$ arguments each, and one n-ary function f, then we can combine them to obtain a new function h of k arguments as follows:

$$h(x_1, \ldots, x_k) = f(g_1(x_1, \ldots, x_k), \ldots, g_n(x_1, \ldots, x_k)). \quad (I)$$

(2) RECURSION. Let f and g be two *total* functions of k and $k + 2$ arguments, respectively. Then a function h of $k + 1$ arguments is said to be obtained from f and g by recursion iff the following equations hold :

$$h(0, x_1, \ldots, x_k) = f(x_1, \ldots, x_k)$$

$$h(m + 1, x_1, \ldots, x_k) = g(m, h(m, x_1, \ldots, x_k), x_1, \ldots, x_k)$$

The simplest case of course occurs when $k = 0$, in which case f is some constant number c. The above equations then become

$$h(0) = c$$

[6] Much as a logical formula is built up from one or more "atomic sentences" by a finite number of applications of the well-known "formation rules".

$$h(m+1) = g(m, h(m))$$

Note that from our stipulation that f and g be total it follows that we cannot obtain non-total functions by means of recursion.

A function which can be obtained from the initial functions by a finite number of applications of composition and recursion is called **primitive recursive**. As an example, here is a primitive recursive derivation of binary addition from the successor and projection functions:

$$+(0, x) = p_1^1(x)$$

$$+(m+1, x) = s(p_3^2(m, +(m, x), x))$$

Since $p_1^1(x)$ and $s(p_3^2(m, +(m, x), x))$ are both primitive recursive (the first being an initial function, the second a composition of two initial functions[7]), $+(x, y)$ is primitive recursive as well. Of course since $p_1^1(x) = x$ and $p_3^2(m, +(m, x), x) = +(m, x)$, we can (informally) write the above equations in the following manner:

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

$$+(m+1, x) = s(+(m, x))$$

And if we use infix notation and write $s(x)$ as $x + 1$ we can simplify things even further:

$$0 + x = x$$

$$(m+1) + x = (m + x) + 1$$

It is not difficult to see that all primitive recursive functions are Pascal-computable. The initial functions certainly are. And in the composition schema (I), if f and the g_is are Pascal-computable then so is their composition. In particular, let P and $\mathsf{P_1}, \ldots, \mathsf{P_n}$ be the subroutines that compute f and $g_1, \ldots, g_n$, respectively. Then the following subroutine computes $h(x_1, \ldots, x_k) = f(g_1(x_1, \ldots, x_k), \ldots, g_n(x_1, \ldots, x_k))$:

```
FUNCTION Compose(x_1,...,x_k : INTEGER) : INTEGER;
VAR
temp1,..., tempn : INTEGER;
BEGIN
    temp1 := P1(x_1,...,x_k);
        ⋮
    tempn := Pn(x_1,...,x_k);
```

[7]Note that the composite function $s(p_3^2(x_1, x_2, x_3))$ is actually a function of three arguments, not one (it can be written as $g(x_1, x_2, x_3) = s(p_3^2(x_1, x_2, x_3))$).

```
    Compose := P(temp1,...,tempn);
END;
```

If a g_i ($i \in \{1, \ldots, n\}$) is undefined for some input arguments $x_1, \ldots, x_k$, then (by definition of Pascal-computability) Pi(x_1,...,x_k) will get into an infinite loop, and thus the statement tempi := Pi(x_1,...,x_k) will never terminate and Compose will diverge (appropriately, since $h(x_1, \ldots, x_k) \uparrow$). The same thing happens if f is undefined for temp1, ..., tempk. On the other hand, if both $g_i(x_1, \ldots, x_k) \downarrow$ (for all $i = 1, \ldots, n$) and $f(g_1(x_1, \ldots, x_k), \ldots, g_n(x_1, \ldots, x_k)) \downarrow$, then both P and the Pis converge, and so does Compose (again appropriately, since in that case $h(x_1, \ldots, x_k) \downarrow$). Further, if Compose does converge, it evidently outputs the correct value. Similar reasoning can show that if two (total) functions f and g are Pascal-computable, then so is any function obtained from f and g by recursion.

In light of the fact that Pascal-computable functions are (trivially) mechanically computable, what we just showed in the previous paragraph is that all primitive recursive functions are mechanically computable. How about the converse? Are all mechanically computable functions primitive recursive? That would entail that

Mechanical computability = Primitive recursiveness. (A)

That is not the case. The simplest refutation we can give is the fact that *only total functions* are primitive recursive (since (1) the initial functions are total, and (2) if we start with total functions then composition and recursion can only yield new total functions). Hence all the non-total functions which we intuitively judge computable are *a priori* ruled out, thereby precluding the truth of eq. (A).

But one might still wonder what the case is if we restrict our attention to total functions proper. Is mechanical computability to be identified with primitive recursiveness then? Well, the vast majority of the algorithmically computable total functions one encounters "in practice" *are* primitive recursive. Earlier we showed that the addition function is primitive recursive, for instance. In similar ways one can prove that bounded subtraction[8], multiplication, (integer) division, exponentiation (hence all nt polynomials), the factorial function, and many others, are all primitive recursive. So at least empirical evidence would seem to suggest that in the domain of total functions,

[8] The *total* function $x \dot{-} y = \begin{cases} x - y & \text{if } x > y \\ 0 & \text{otherwise.} \end{cases}$

mechanical computability and primitive recursiveness are co-extensive. In fact in the early days of computability several researchers (most notably Herbard) believed that to be the case. But it is not. An early counter-example was adduced by Ackermann. Define a function $A : N^2 \longrightarrow N$ as follows:

$$A(0, n) = n + 1$$
$$A(m + 1, 0) = A(m, 1)$$
$$A(m + 1, n + 1) = A(m, A(m + 1, n)).$$

It can be proved that A is not primitive recursive.[9] Yet it is obvious that its values can be computed algorithmically (one could easily write a Pascal function for that purpose). Actually, *because* all primitive recursive functions are total there is an easier (albeit more ad hoc) way to produce a counter-example: diagonalization. Appendix A gives one an idea of how such a construction would proceed.

It turns out that if we extend the class of primitive recursive functions by adding the operation of unbounded minimalization to composition and recursion, then we do obtain *all* the mechanically computable functions, both total and not. This third definitional schema can be formulated as follows:

UNBOUNDED MINIMALIZATION. Let g be a function of $k + 1$ arguments. Then we say that a function $f : N^k \longrightarrow N$ is obtained from g by unbounded minimalization iff for all $x_1, \ldots, x_k$,

$$f(x_1, \ldots, x_k) = min_y[g(x_1, \ldots, x_k, y) = 1 \ \wedge \ (\forall z \leq y)\ g(x_1, \ldots, x_k, z) \downarrow].$$

So $f(x_1, \ldots, x_k)$ is the smallest number y such that $g(x_1, \ldots, x_k, z)$ is defined for all $z = 0, 1, \ldots, y$ *and* $g(x_1, \ldots, x_k, y) = 1$— *if such a y exists.* If not, then $f(x_1, \ldots, x_k)$ is undefined. This immediately engenders the possibility that f might turn out non-total *even if g is total.* For example, define $g : N^3 \longrightarrow N$ as

$$g(x_1, x_2, x_3) = \begin{cases} 1 & \text{if } x_2 + x_3 = x_1 \\ 0 & \text{otherwise.} \end{cases}$$

g is the characteristic function of a relation on N^3, that which obtains whenever x_3 is the difference $x_1 - x_2$ (e.g $g(8, 5, 3) = 1$, but $g(5, 8, 3) = 0$). Obviously, g is total. Nevertheless, the function

$$f(x_1, x_2) = min_y[g(x_1, x_2, y) = 1 \text{ and } (\forall z \leq y)\ g(x_1, x_2, z) \downarrow]$$

[9]The reason being that it grows too fast, much faster than primitive recursiveness would allow. Specifically, one can easily prove that for all primitive recursive functions f there is a constant number c such that $A(c, max(x, y)) > f(x, y)$ for all x and y.

is *not* total. For, a careful inspection of the minimalization will show that what we have obtained is the function

$$f(x_1, x_2) = \begin{cases} x_1 - x_2 & \text{if } x_1 \geq x_2 \\ \uparrow & \text{if } x_1 < x_2, \end{cases}$$

which is obviously non-total.

We finally define a function to be **recursive** iff it can be obtained from the initial functions by a finite number of applications of composition, recursion, and minimalization.

3.3 Third Formalization: Turing machines

A **Turing machine** M (fig. 3.1) consists of three parts:
(1) An one–way infinite tape of square cells. The tape has a left end but extends indefinitely to the right. Each square has either a blank (printed as '#') or an '1' on it.
(2) A "processor" with a read/write head which can scan only one square at a time (the "current square"). The processor can be in any one of a finite number of states $Q_M = \{q_0, q_1, \ldots, q_m\}$, $m > 0$.
(3) A finite list of instructions $I_1, \ldots, I_n$, $n \geq 0$, called **the program of M** and denoted as P_M. Each instruction is a five-tuple of the form $q\ s\ q'\ s'\ A$, where $q, q' \in Q_M$, $s, s' \in \{1, \#\}$, and $A \in \{L, R\}$. A typical instruction is $q_2\ 1\ q_5\ \#\ L$. The interpretation is that if the processor is in state q_2 and the current square contains an 1, then the processor changes to state q_5, the read/write head writes the symbol # on the current square (overwriting the 1 that was previously there), and then it moves one square to the left.

We now present the "run–time semantics" of Turing machines. Suppose that in the course of a certain computation, a machine M has reached a state q_i, $i \neq 0$, and that the symbol on the current square is s (we will shortly explain how computations get started). Then either (i) P_M contains one or more instructions $I_{j_1}, \ldots, I_{j_p}$, $1 \leq j_1 < j_2 < \ldots < j_p \leq n$ of the form $q_i\ s\ q_j\ s'\ A$ (for some $q_j \in Q_M, s' \in \{1, \#\}, A \in \{L, R\}$), or (ii) not.

- If (i) is the case then the *first* [10] instruction I_{j_1} of the form $q_i\ s\ q_j\ s'\ A$ is executed, which means that
  (a) M changes to state q_j,
  (b) the symbol s on the current square is overwritten with s', and

[10] Thus even when there are several choices ($p > 1$), computation always proceeds deterministically with the unique "topmost" instruction,

Figure 3.1: A Turing machine.

(c) the read/write head moves one square to the left or one to the right, depending on the value of A— *unless* the current square is the leftmost and $A = L$, in which case there is no movement, i.e the head stays on the same (leftmost) square.

If the new state q_j is the "halting state" q_0 then all activity stops and we say that M has halted. In that case we define the **output** of the computation to be the total number of 1s left on the tape.

- If (ii) is the case then the computation cannot proceed any further and we again say that M halts and that its output (for that particular computation) is the total number of 1s left on the tape.

Finally, a computation begins by moving the read/write head to the leftmost square and putting M in the "starting state" q_1. Execution of P_M then proceeds in accordance with the above stipulations.

We are now ready to make the connection with pnt functions. A number x is given to M as **input** by writing $x+1$ consequtive 1s on the $x+1$ leftmost squares of the tape (thus a zero argument is represented by a single 1 on the first square). The rest of the tape contains blank symbols # throughout. Multiple input arguments are sequentially represented in the same fashion, with exactly one blank separating each input number. Specifically, we supply

M with $k > 0$ input arguments $x_1, \ldots, x_k$ by writing $x_1 + 1$ consequtive 1s on the left end of the tape, then a blank # on the $(x_1 + 2)^{nd}$ square, then $x_2 + 1$ consequtive 1s after that, and so forth. We again assume that after the last $x_k + 1$ 1s the tape contains blank symbols throughout.

Now for any $k > 0$ numbers $x_1, \ldots, x_k$, we define $\psi_M^k(x_1, \ldots, x_k)$ to be the output of M when the numbers $x_1, \ldots, x_k$ are supplied as input to M in the manner described above, M is put in the starting state q_1, the read/write head is moved to the leftmost square, and the processor begins executing the program P_M in accordance with the aforementioned rules. If the computation ever stops then we say that **M halts** (or **"converges"**) on input(s) $x_1, \ldots, x_k$, and we write $\psi_M^k(x_1, \ldots, x_k) \downarrow$. In that case $\psi_M^k(x_1, \ldots, x_k)$ is the output of the computation, that is, the total number of 1s left on the tape after M halts. On the other hand, if the computation continues *ad infinitum* then we say that M **diverges** on inputs $x_1, \ldots, x_k$ and we write $\psi_M^k(x_1, \ldots, x_k) \uparrow$, to indicate that the value $\psi_M^k(x_1, \ldots, x_k)$ is undefined. In the case of a single input argument the superscript $k = 1$ is dropped and we simply write $\psi_M(x)$.

It is noteworthy that $\psi_M^k(x_1, \ldots, x_k)$ is a well-defined pnt function for *all* $k > 0$. For even though M might be designed to work only with a specific number c of input arguments, still, if we supply M with *any* number of arguments $x_1, \ldots, x_k$, even if $k \neq c$, and we start executing P_M, M will necessarily either halt, in which case $\psi_M(x_1, \ldots, x_k)$ is a well-defined number, or diverge, in which case $\psi_M(x_1, \ldots, x_k)$ is undefined. Hence, although M might be *intended* to work and produce sensible results only with one particular number of arguments, it nevertheless determines a countable infinity of functions $\{\psi_M^k : N^k \longrightarrow N\}_{k=1}^{\infty}$.

Finally, let f be any pnt function of $k > 0$ arguments. We say that a Turing machine M **computes** f iff

$$\text{for all numbers } x_1, \ldots, x_k, \quad f(x_1, \ldots, x_k) = \psi_M^k(x_1, \ldots, x_k),$$

meaning that
(1) $\psi_M^k(x_1, \ldots, x_k) \uparrow$ iff $f(x_1, \ldots, x_k) \uparrow$, and
(2)if $f(x_1, \ldots, x_k) \downarrow$, then $\psi_M^k(x_1, \ldots, x_k) \downarrow$ and $f(x_1, \ldots, x_k) = \psi_M^k(x_1, \ldots, x_k)$.
In other words, M computes f iff it diverges on all and only those input numbers $x_1, \ldots, x_k$ for which f is undefined, and it halts with output $f(x_1, \ldots, x_k)$ for all input numbers $x_1, \ldots, x_k$ for which f is defined. Accordingly, we say that a pnt function is **Turing–computable** iff there is a Turing machine that computes it.

3.4 Equivalence of the formalizations

As we have already emphasized, all three formalizations turn out to be extensionally equivalent. In other words, one can prove:
THEOREM 3.4.1. *A pnt function is Pascal-computable iff it is recursive iff it is Turing-computable.*

There are several ways to prove this theorem. One could, for example, show first that (1) Pascal-computability = recursiveness, and then that (2) recursiveness = Turing-computability. Transitivity would then also equate Pascal-computability and Turing-computability. One half of (1) and (2), showing that recursiveness implies Pascal- or Turing-computability, is easy. We have already shown (section 3.2) that all recursive functions are Pascal-computable. In a similar manner we could show that they are also Turing-computable. In the other direction of (1) and (2), i.e in showing that Turing- (or Pascal-) computability implies recursiveness, we use arithmetization to encode Turing machines (or Pascal programs) and their convergent computations by numbers. Then we invoke Kleene's **normal form theorem**, adopted to the particular formalism at hand. In the case of Turing machines, for instance, the theorem asserts that for all numbers i, k $(k > 0)$, there is a primitive recursive relation $T \subseteq N^{k+2}$ and a primitive recursive function $U : N \longrightarrow N$ such that

$$\psi_{M_i}(x_1, \ldots, x_k) = U(min_y[T(i, x_1, \ldots, x_k, y)]). \quad (I)$$

Intuitively, $T(i, x_1, \ldots, x_k, y)$ holds iff number y is the code of a (convergent) computation of machine M_i on inputs $x_1, \ldots, x_k$. Thus $min_y T(i, x_1, \ldots, x_k, y)$ picks one such computation (the one with the smallest code number), and then the function U simply outputs the total number of 1s left on the tape at the end of computation $\#y$. The details are tedious, and T and U turn out quite monstrous-looking. Nonetheless, they are primitive recursive, and that is all we need to conclude, from equation (I), that every Turing-computable function $\psi_M : N^k \longrightarrow N$ is recursive.

Another way to prove theorem 3.4.1 is to show that (3) Pascal-computability entails recursiveness, (4) recursiveness entails Turing-computability, and, closing the circle, (5) Turing-computability entails Pascal-computability. We have already indicated how the first two steps would be performed: for (3) we would use an adaptation of the normal form theorem (we would have to encode Pascal subroutines and their computations as numbers for that); while for (4) we would show that the initial functions are Turing-computable and that functions obtained from Turing-computable functions via composition, recursion, and unbounded minimalization are also Turing-computable.

For (5) we would have to simulate Turing machines with Pascal subroutines, which is the standard approach for proving the equivalence of two distinct models of computation.

At any rate, the technicalities of the proof are not that important. More important is the theorem's content, the assertion it makes. It implies that either all three formalizations are accurate or none is; either they all hold up or they all go down together. For, if one adduced a pnt function f that is mechanically computable on intuitive grounds but not, say, Turing-computable, then the theorem would entail that f is also Pascal-uncomputable and non-recursive. Thus all three definitions would be proven under-inclusive. The existence of such a counter-example is considered to be extremely unlikely, for reasons that we discuss in the following section.

3.5 Church's thesis and its practical significance

Church's thesis is the assertion that Turing's formalization is accurate. Of course, from what we said in the previous section, that is equivalent to asserting the accuracy of either one of the other two definitions. Indeed, we could have propounded Church's thesis in terms of, say, recursive functions. We chose Turing's formalization only because it is the one we will be using in the sequel.

In section 1.3 we analyzed statements such as Church's thesis (that assert the accuracy of a mathematical definition) as universally quantified empirical propositions. We therefore concluded that, although they can be definitively falsified (by means of counter-examples), they can never be definitively established. The best we can hope for in the way of substantiative evidence is the accumulation of greater and greater numbers of corroborating instances. As we keep discovering individual cases that uphold the definition, our confidence in its accuracy grows stronger. The greater the number of such cases, the stronger our confidence. But no matter how improbable it might come to be considered, a counter-example will always be at least a theoretical possibility; it cannot be ruled out on *a priori* grounds.

That is exactly how things are with Church's thesis. The main reason for its almost universal acceptance is the huge amount of validating evidence that has accrued over the decades. A tremendous number of functions which are algorithmic in the intuitive sense have proved to be Turing-computable as well.[11] By contrast, and in spite of many zealous attempts, not a single

[11] I will follow the convention of regarding the other half of the thesis, the assertion that Turing-computable functions are mechanically computable, as patently and thus

counter-example has been discovered so far. Therefore, probabilistically, the odds are unwaveringly in favor of Church's thesis.

But inductive proof is not the only bolster supporting the thesis. In chapter 1 we maintained that intuition and pragmatic considerations are also key factors in the process of accepting a mathematical definition as accurate. Both are on the side of Church's thesis. Most people with some programming experience and a fair understanding of the concept of mechanical computability are apt to "see" that the said concept cannot amount to anything more than Pascal-computability (or equivalently, Turing-computability). It is the same type of intuition (Plato would call it our mathematical "sixth sense") that makes one "see" the accuracy of Tarski's definition of truth or Weirstrass's definition of limits. And pragmatic considerations play a role, too. The endorsement of Church's thesis yields sensible, intuitively appealing results. And then we also have theorem 3.4.1, which can actually be extended to include many other formalizations, such as Post systems (Markov algorithms), flowchart languages, the λ-calculus, equational systems, RAM programs, C programs, LISP programs, or, indeed, any programming language of the customary kind. How can the thesis be wrong when each of these formalizations seems accurate and they all turn out co-extensive? At any rate, even in the highly unlikely event that the thesis is false, the class of pnt functions that is picked out by all these definitions *must* be very "natural" and computationally important— and that alone would make its study worthwhile.

Now the practical significance of Church's thesis for our purposes will be this: whenever we manage to come up with an algorithm that computes a certain function f, we will go ahead and conclude without further ado that f is Turing-computable. For, the existence of the algorithm obviously means that f is mechanically computable; and the thesis assures us that all such functions are Turing-computable ("if something can be done mechanically, it can be done by a Turing machine"). That will save us a great deal of tedious work. It will be analogous to working with a neat high-level language like Pascal versus working with a machine language and dealing with individual 0s and 1s. Instead of fastidiously constructing Turing machines down to the last painstaking detail, we will give informal (but precise) algorithms in a "high-level" mixture of English and mathematical parlance. As long as our algorithm is legitimate (rigorous, deterministic, etc.), Church's thesis will warantee that it can be carried out by some Turing machine.

---

uncontroversially true.

Chapter 4

Mechanical unsolvability

4.1 Encoding techniques

In this section we present methods for encoding and decoding (i) pairs of numbers, and more generally, (ii) arbitrarily long finite sequences of numbers. We begin with the former.

To encode pairs of numbers we need a bijective "pairing function" $P : N^2 \longrightarrow N$ that will associate a unique number z to every pair (x, y) and a unique pair (x, y) to every number z. P will be the encoding function. We will also need two decoding functions $l : N \longrightarrow N$ and $r : N \longrightarrow N$ such that $l(z)$ and $r(z)$ will be the left and right elements, respectively, of the pair encoded by z. In other words, for all x and z we will have $l(z) = x$ iff there is a y such that $P(x, y) = z$; and for all y and z we will have $r(z) = y$ iff there is an x such that $P(x, y) = z$.

We derive the desired functions from Cantor's method for enumerating the rational numbers.[1] His idea was to arrange all pairs of numbers in a rectangular table that has an upper left end but extends indefinitely in all other directions, as shown in fig. 4.1, and then start traversing the table in the manner indicated by the arrows: first we visit (0,0), then (0,1), then (1,0), then (0,2), (1,1), (2,0), (0,3), and so on. This enumeration precedure will eventually generate any given pair of numbers (x, y). Therefore, we can simply set

$P(x, y) = n$ iff (x, y) is the n^{th} pair to be enumerated by this method.

Defined in this manner, P is both one-one and onto. Moreover, we can easily find a closed formula for $P(x, y)$ by counting the numbers of pairs

[1] Which, of course, are nothing but pairs of numbers.

Figure 4.1: Cantor's method for putting the rationals in an one–one and onto correspondence with the natural numbers.

enumerated before (x, y). We do that as follows.

First we note that all pairs on the same diagonal connection have identical sums. In particular,
(i) the sum of each pair on the i^{th} diagonal is i. Therefore, we know that a pair (x, y) is located somewhere on the $(x + y)^{th}$ diagonal. Further,
(ii) For all $i \geq 0$, there are exactly $i + 1$ pairs on the i^{th} diagonal.
Now, given an arbitrary pair (x, y), how many pairs are enumerated before (x, y)? Answer: (a) the number of pairs on previous diagonals, plus (b) the number of previous pairs on the same diagonal. Since, by (i), (x, y) is on the $(x + y)^{th}$ diagonal, quantity (a) is the number

$$\sum_{i=0}^{x+y-1} [\#\text{ of pairs on the } i^{th} \text{ diagonal}], \quad \text{which, by (ii), is } \sum_{i=0}^{x+y-1} i + 1.$$

For (b), we observe that for two pairs (x, y) and (x', y') *on the same diagonal,* (x, y) precedes (x', y') iff $x < x'$. Hence, number (b) is simply x. Thus we conclude that

$$\text{(a) + (b)} = x + \sum_{i<x+y} i + 1 \;=\; x + \sum_{i<x+y} i + \sum_{i<x+y} 1 \;=$$

$$2x + y + \frac{(x + y - 1)\,(x + y)}{2} \;=\; \frac{3x + y + (x + y)^2}{2}.$$

Therefore, we can now formally define

$$P(x, y) = \frac{3x + y + (x + y)^2}{2}.$$

Note that for all x and y, $x \leq P(x, y)$ and $y \leq P(x, y)$.

We can now also define the decoding functions

$$l(z) = min_{x \leq z}[(\exists y \leq z)\,(z = P(x, y))]$$

$$r(z) = min_{y \leq z}[(\exists x \leq z)\,(z = P(x, y))].$$

These are clearly mechanically computable: to compute $l(z)$, or $r(z)$, we simply start enumerating pairs of numbers in the manner described above

until we generate the z^{th} pair (x, y); then we simply let $l(z) = x$ and $r(z) = y$. We summarize the properties of P, l, and r as follows:
(1) $l(P(x, y)) = x,\ \ r(P(x, y)) = y$
(2) $P(l(z), r(z)) = z$
(3) $x, y \leq P(x, y)$

We now move on to arbitrarily long finite sequences of numbers. We define an encoding function $S : \bigcup_{i>0} N^i \longrightarrow N$ by letting $S(x_1, \ldots, x_k)$ be the number (written in binary notation)

$$1\ \overbrace{0\cdots0}^{x_1 \text{ times}}\ 1\ \overbrace{0\cdots0}^{x_2 \text{ times}}\ 1\cdots1\ \overbrace{0\cdots0}^{x_k \text{ times}}\ .$$

E.g $S(2, 3) = 1001000 = 72, S(1, 4, 2) = 1010000100 = 44, S(3, 0, 2) = 10001100 = 140, S(0, 0) = 11 = 3, S(0) = 1$, etc. More formally,

$$S(x_1, \ldots, x_k) = \sum_{i=1}^{k} 2^{(x_i+\ldots+x_k)+k-i} \ = \sum_{i=1}^{k} 2^{k-i+\sum_{j=1}^{k} x_j}.$$

S is one–one because two different sequences will result in two different bit strings— which, as binary numbers, will have distinct values. S is also onto, because every number $n > 0$ can be represented as a binary string whose leftmost bit is 1; and, according to our stipulations, every such string encodes a unique sequence of numbers. For example, the string 100110 encodes the sequence [2,0,1]. The only number that does not encode a sequence is zero. Therefore, it is convenient to regard zero as encoding the "empty sequence".

We now define a "length function" $LEN : N \longrightarrow N$ as

$$LEN(n) = \textit{the length of the sequence encoded by } n.$$

E.g, since 3 is the code number of [0,0] (in binary 3 is 11), we have $LEN(3) = 2$. Likewise, $LEN(255) = 8$, since $255 = 11111111$ encodes the 8–member sequence [0,0,0,0,0,0,0,0], $LEN(16) = 1$, etc. In general, $LEN(n) = k$ iff there are k (unique) numbers $x_1, \ldots, x_k$ such that $S(x_1, \ldots, x_k) = n$. Note that $LEN(n) > 0$ iff $n > 0$. Formally, we can define LEN primitive recursively as

$$LEN(n) \ = \sum_{i=0}^{\lfloor \log n \rfloor + 1} (n/2^i)\ mod\ 2$$

where, as is well-known, $\lfloor \log n \rfloor + 1$ is the length of the binary representation of n. The above formula simply sums up the number of 1s in the binary

representation of n. That number is the length of the sequence encoded by n.

We also define a family of decoding functions $D_i : N \longrightarrow N$, for each $i > 0$, such that $D_i(n)$ returns the i^{th} element of the sequence encoded by n, *if* that sequence contains $\geq i$ elements. Otherwise the value of $D_i(n)$ is immaterial, and here we arbitrarily stipulate that it be zero. Thus if $S(x_1, \ldots, x_k) = n$, then for all $i \in \{1, \ldots, k\}$ we have $D_i(n) = x_i$, while for $i > k$ we have $D_i(n) = 0$. For example, since $290 = 100100010 = S(2, 3, 1)$, we have $D_1(290) = 2$, $D_2(290) = 3$, $D_3(290) = 1$, and $D_i(290) = 0$ for all $i > 3$. Given arbitrary i and n, the value $D_i(n)$ can be computed mechanically as follows: first we check to see whether i is in the correct range $\{1, \ldots, LEN(n)\}$. If it is not, we indifferently output a 0 and we are done. Otherwise we convert n to binary and we output the number of adjacent 0s immediately to the right of the i^{th} 1 (counting from left to right). But we can also derive a closed formula for $D_i(n)$, $1 \leq i \leq LEN(n)$, by defining a function $LAST : N \longrightarrow N$ as $LAST(n) =$ *the number of adjacent 0s at the right end of the binary representation of* n. E.g $LAST(20) = 2$, since 20 is 10100, $LAST(3) = 0$ since 3 is 11, etc. Therefore, if $S(x_1, \ldots, x_k) = n$, $LAST(n) = x_k$. Formally, we have

$$LAST(n) = min_{k \leq \lfloor \log n \rfloor + 1}[\ (n/2^k)\ mod\ 2 = 1\ ].$$

Now let $S(x_1, \ldots, x_k) = n$. We set

$$D_i(n) = LAST(n), \text{ for } i = k, \text{ and}$$

$$D_i(n) = LAST(\ n/2^{k-i+\sum_{j=i+1}^{k} D_j(n)}\ ) \text{ for } 1 \leq i \leq k.$$

Lastly, but very importantly: in order to conform with the notation used by most contemporary texts on these subjects, we will consistently write
(1) $< x, y >$ instead of $P(x, y)$,
(2) $[x_1, \ldots, x_k]$ instead of $S(x_1, \ldots, x_k)$, and
(3) $(n)_i$ instead of $D_i(n)$.

4.2 Arithmetizing Turing machines

We are now ready to *arithmetize* Turing machines, i.e to encode them by numbers. Our scheme will be one–one and onto, i.e it will assign a unique number $\#(M)$ to each machine M, and it will make sure that every number gets assigned to some machine. In addition, the encoding and the decoding will be mechanically performeable: we will give simple algorithms (actually,

formulae) both for computing the code number of a given machine (encoding) and for retrieving the machine encoded by a given number (decoding).

What we will actually encode are the programs of Turing machines, not Turing machines *per se*, as we have defined them (i.e as physical devices comprising a "work tape", a "processor", etc.). Of course this is perfectly legitimate because from a formal standpoint a Turing machine M *is* its program P_M, in the sense that if we are given P_M we have all the information we will ever need: we can then go ahead and perform any computation with M we want, for any input number(s). The processor, the tape, and the other components of the empirical apparatus of M are little more than mere accessories; they are necessary not mathematically but only intentionally, insofar they assist us humans in relating to and understanding the computational nature of the machine. At any rate they are "standard equipment" that all Turing machines have. The one and only thing that differentiates a particular machine M from all the other machines is its program P_M. Hence from now on we will use the terms 'Turing program' and 'Turing machine' interchangeably. We proceed with the details of the encoding.

Recall that the program P_M was defined as a finite list of instructions $I_1, , \ldots, I_n,\ \ n \geq 0$,[2] where each instruction is a quintuple of the form $q_i\ \ s\ \ q_j\ \ s'\ \ m$ such that $0 \leq i, j\ \leq \mid Q_M \mid,\ \ s, s' \in \{1, \#\}$, and $m \in \{L, R\}$. We begin by defining a finite bijection $C : \{1, \#\} \times \{1, \#\} \times \{L, R\} \longrightarrow \{0, 1, \ldots, 7\}$ by means of the following table:

s	s'	m	$C(s, s', m)$
#	#	L	0
#	#	R	1
#	1	L	2
#	1	R	3
1	#	L	4
1	#	R	5
1	1	L	6
1	1	R	7

This function assigns a unique code number (from 0 to 7) to every possible configuration of $s,\ s'$ and m in an instruction $q_i\ \ s\ \ q_j\ \ s'\ \ m$. For the instruction $q_3\ \#\ q_5\ 1\ L$, for example, we have $C(\#, 1, L) = 2$. We also define three decoding functions $First : \{0, \ldots, 7\} \longrightarrow \{1, \#\}$, $Second : \{0, \ldots, 7\} \longrightarrow \{1, \#\}$, and $Third : \{0, \ldots, 7\} \longrightarrow \{L, R\}$, in the obvious

[2]For $n = 0$ we get the empty program, which contains no instructions.

way (so that $First(5) = 1$, $Second(5) = \#$, $Third(5) = R$, $First(2) = \#$, etc.). Next we define the **godel number of an instruction** $I = q_i\ s\ q_j\ s'\ m$ to be the number

$$\#(I) = < i, (8 * j) + C(s, s', m) > .$$

The godel number of $I = q_1\ 1\ q_0\ \#\ L$, for instance, is $\#(I) =< 1, (8 * 0) + C(1, \#, L) > = < 1, 4 >= 16$, while for $I = q_5\ \#\ q_9\ \#\ R$ we have $\#(I) = < 5, 72 + C(\#, \#, R) > = < 5, 73 > = 3086$.

Observe that if two instructions $I = q_i\ s\ q_j\ s'\ m$ and $I' = q_\alpha\ \sigma\ q_\beta\ \sigma'\ m'$ are not identical (i.e if they differ in at least one of their five components), then either $i \neq \alpha$, or $(8 * j) + C(s, s', m) \neq (8 * \beta) + C(\sigma, \sigma', m')$. Therefore, since the pairing function $< x, y >$ is one-one, the numbers $< i, 8 * j + C(s, s', m) > = \#(I)$ and $< \alpha, 8 * \beta + C(\sigma, \sigma', m') > = \#(I')$ must be distinct, which proves that different instructions have different godel numbers. It is equally easy to show that every number encodes some instruction. Furthermore, given a number n, we can easily retrieve the instruction $I = q_i\ s\ q_j\ s'\ m$ encoded by it: we have $i = l(n)$, $j = r(n)/8$, and since $C(s, s', m) = r(n)\ \ mod\ \ 8$, we have $s = First(r(n)\ \ mod\ \ 8)$, $s' = Second(r(n)\ \ mod\ \ 8)$, and $m = Third(r(n)\ \ mod\ \ 8)$. For example, 2 is the godel number of the instruction $q_1\ \#\ q_0\ \#\ L$, because $2 = < 1, 0 > = < 1, (8 * 0) + 0 >$, while 1 and 0 are the godel numbers of the instructions $q_0\ \#\ q_0\ \#\ R$ and $q_0\ \#\ q_0\ \#\ L$, respectively. As a last example, 5893 is the godel number of $q_7\ 1\ q_{12}\ \#\ R$.

Finally, we define the **godel number of a Turing machine** M to be

$$\#(M) = [\#(I_1), \ldots, \#(I_n)]$$

where $I_1, \ldots, I_n$ are the instructions of P_M. Note that if P_M is the empty program ($n = 0$), then $\#(I_1), \ldots, \#(I_n)$ is the empty sequence, so in virtue of the convention we made in the previous section, $[\#(I_1), \ldots, \#(I_n)] = 0$. Appropriately, then, 0 is the godel number of the empty program.

Since the encoding function $[x_1, \ldots, x_k]$ is one–one, different programs have different godel numbers. Moreover, since (1) for any given number $n > 0$ there are $k > 0$ numbers $x_1, \ldots, x_k$ such that $[x_1, \ldots, x_k] = n$, and (2) each x_i $(i = 1, \ldots, k)$ is the godel number of some instruction, it follows that every n is the godel number of some Turing program, i.e that our encoding is onto as well as one–one.[3] For example, 1 is the godel number of the Turing program whose sole instruction I is $q_0\ \#\ q_0\ \#\ L$, since

[3]Note that our encoding of Turing machines would *not* be onto if our encoding of instructions was not onto as well, because then premise (2) would not necessarily be true.

$\#(I) = < 0, 0 > = 0$, so that $[\#(I)] = [0] = 1$. Finally, we write M_x to denote the unique Turing machine encoded by number x. Thus the list

$$M_0,\ M_1,\ M_2,\ M_3,\ \ldots$$

is an enumeration of all Turing machines (every machine will appear exactly once in this list, its exact position depending on the size of its godel number). Also, we write P_x for the program of M_x (again, P_x and M_x will be used synonymously). And, as should be expected, we write $\psi^k_{M_x}(x_1, \ldots, x_k)$ for the k–argument function computed by M_x (see section 3.3).

4.3 The Universal Turing machine

Our main objective in this section will be to answer the following question : **Is it possible to design a "Turing interpreter", i.e a Turing program that "runs" arbitrary Turing programs?**
We can certainly do this type of thing with every–day programming languages. We can write a LISP interpreter in LISP, for example. Such an interpreter would be a LISP program that takes as input an arbitrary LISP program and runs it, or better yet, *simulates* it. Now in our case the question is whether there exists a Turing machine that will take as inputs
(1) an arbitrary Turing machine M, and
(2) an arbitrary number x,
and will start executing the program of M on input x.[4]

Let us make things a little more formal. Let us define a function $\Phi : N \longrightarrow N^2$ as follows:

$$\Phi(x, y) = \psi_{M_y}(x).$$

In other words, the value of $\Phi(x, y)$ will be the output produced when we run M_y on input x. Suppose, for example, that M_{358} is a Turing machine that computes the successor function. Then $\Phi(9, 358) = \psi_{M_{358}}(9) = 10$, $\Phi(999, 358) = 1000$, etc. Of course Φ is not total because for some x and y, M_y diverges on x, i.e $\psi_{M_y}(x) \uparrow$ and thus $\Phi(x, y) \uparrow$. For example, if 2157 is the godel number of a machine that diverges on all odd inputs x, then $\Phi(31, 2157) \uparrow$, $\Phi(5, 2157) \uparrow$, and, in general, $\Phi(2x + 1, 2157) \uparrow$ for all numbers x.

Φ is essentially a formalization of our interpreter problem. If Φ is Turing–computable, i.e if there is a Turing machine M_{int} that computes Φ, then our

[4]For now we restrict our attention to one input number for the sake of simplicity. Later it will be easy to extend things and cover cases in which M takes several arguments.

question has an affirmative answer: M_{int} will be the "Turing interpreter" we are looking for. We can therefore rephrase our original question in the following manner:

Is the function $\Phi(\mathbf{x}, \mathbf{y})$ Turing–computable? In other words, is there a Turing machine $\mathbf{M_{int}}$ which, given arbitrary numbers x and y, will

(1) converge and output the value $\psi_{\mathbf{M_y}}(\mathbf{x})$ if $\mathbf{M_y}$ halts on x, or

(2) diverge if $\mathbf{M_y}$ does not halt on x?

This question can be answered very easily if we take advantage of Church's thesis: if we manage to discover an informal algorithm for computing Φ, then the thesis will assure us that Φ is Turing–computable. Now an informal algorithm for computing $\Phi(x, y)$ is very easy to give; the following will do the job:

STEP 1: Given the input numbers x and y, use the decoding method described in the previous section to retrieve P_y, the program of M_y.

STEP 2: Write $x + 1$ consequtive 1s on the left end of a blank work tape.

STEP 3: Set $i = 1$ and let the current square be the leftmost one.

STEP 4: Execute the topmost instruction of P_y having the form $q_i\ s\ q_j\ s'\ m$, where s is the symbol written on the current square, *if* there is such an instruction (if not, halt and let the output be the total number of 1s left on the tape). As usual, executing the said instruction amounts to overwriting s with s', changing states from q_i to q_j, and letting the square to the left or right of the current square (depending on the value of m) be the new current square. Take care of special cases (like trying to move to the left of the leftmost square) in accordance with the rules given in the previous section.

STEP 5: If $j = 0$ halt and let the output be the total number of 1s left on the tape. Otherwise, set $i = j$ and goto step 4.

This algorithm simply runs M_y on input x, step by step. If execution ever halts, then the output $\Phi(x, y)$ is the value $\psi_{M_y}(x)$. Otherwise we have $\Phi(x, y) \uparrow$. Thus Church's thesis implies the following important result:

THEOREM 4.3.1. *The function $\Phi(x, y)$ is Turing–computable.*

Of course this means that there is a Turing machine M_{int} that computes Φ. M_{int} is often called the "universal Turing machine", due to its capability to simulate any (one–argument) computation of any machine.

Next we discuss the case of multiple arguments. For each $k > 0$ we define a function Φ^k of $k + 1$ arguments as follows:

$$\Phi^k(x_1, \ldots, x_k, y) = \psi_{M_y}(x_1, \ldots, x_k).$$

In other words, $\Phi^k(x_1, \ldots, x_k, y)$ is the output produced (if any) when M_y is

started with inputs $x_1, \ldots, x_k$. To prove that each Φ^k is Turing-computable we give the same argument we gave for the 1–argument case: Φ^k is mechanically computable in the intuitive sense (the same algorithm that was given above can be used to compute any Φ^k simply by adjusting step 2 to the particular k at hand). Therefore, Church's thesis implies that

THEOREM 4.3.2. *For each $k > 0$, the function $\Phi^k(x_1, \ldots, x_k, y)$ is Turing–computable.*

This means that for each $k > 0$ there is a Turing machine M^k_{int} that can simulate the behavior of *any* Turing machine on any k inputs. Each M^k_{int} is said to be a universal Turing machine.

Sometimes we will write $\Phi^k(x_1, \ldots, x_k, y)$ as $\Phi^k_y(x_1, \ldots, x_k)$. Also, for the 1–argument case (which is the case that will concern us most often) we drop the superscript k and simply write $\Phi(x, y)$ and M_{int} (so that $\Phi^1(x, y) = \Phi(x, y) = \Phi_y(x) = \psi_{M_{int}}(x, y)$).

Next, for each $k > 0$ we define the $STEP^k$ predicate[5] as a function from N^{k+2} to $\{0, 1\}$ in the following way:

$$STEP^k(x_1, \ldots, x_k, y, t) = \begin{cases} 1 & \text{if } M_y \text{ halts on } x_1, \ldots, x_k \textit{ in } t \textit{ or fewer steps} \\ 0 & \text{otherwise.} \end{cases}$$

Being a predicate, each $STEP^k$ is a total function: for any $x_1, \ldots, x_k, y$, and t, either M_y halts on $x_1, \ldots, x_k$ in no more than t steps or it does not. Furthermore, we can determine which is actually the case in a mechanical fashion: we simply retrieve the program P_y and we start executing it on inputs $x_1, \ldots, x_k$, keeping track of the number of steps we go through. If the computation converges while we have not taken more than t steps, we halt and output an 1. Otherwise, if after taking t steps the computation has still not halted, we stop and output a zero. Thus Church's thesis entails the following:

THEOREM 4.3.3. *For each $k > 0$, the predicate $STEP^k(x_1, \ldots, x_k, y, t)$ is Turing–computable.*

In this case, too, when $k = 1$ we drop the superscript and simply write $STEP(x, y, t)$.

Finally, for every number n we define the set

$$W_n = \{x \in N \mid \Phi(x, n) \downarrow\}.$$

[5] A predicate P is a subset of N^k, for some $k > 0$. We will often conflate P with its characteristic function, i.e sometimes we will treat P as a relation on N^k and sometimes as a total function from N^k to $\{0, 1\}$.

In other words, W_n is the set of all and only those numbers for which M_n halts.[6] For example, since M_0 is the "empty machine", we have $W_0 = \emptyset$.

4.4 Uncomputable functions

In this section we will begin to reap the benefits of our formalizations. In particular, we are going to utilize the mathematical machinery we have been developing in order to prove with apodeictic certainty the following remarkable fact: that there are many functions that cannot be computed by Turing machines, or, equivalently, that there are many problems that cannot be solved mechanically. Of course the mere existence of Turing–uncomputable functions is not, by itself, an astounding fact. It actually ought to be expected: there is only a scanty countable infinity of Turing machines next to an uncountable infinity of pnt functions. Therefore, it is natural that the vast majority of such functions should fail to be Turing–computable. What is remarkable, however, is that we should have "mental access" to such uncomputable objects; that we should be able to apprehend and define them with precision and thoroughness; and, finally, that we should be able to *prove* them uncomputable. Without further ado, we begin our list of uncomputable functions with the legendary "halting problem":

THEOREM 4.4.1. *The function*

$$SHALT(x) = \begin{cases} 1 & \text{if } \Phi(x,x)\downarrow \\ 0 & \text{if } \Phi(x,x)\uparrow \end{cases} \quad \textit{is not Turing–computable.}$$

Interpretation by Church's thesis: *There is no algorithm for determining whether or not an arbitrary Turing machine halts when started with its own godel number as input.*

Proof. By way of contradiction, assume that there is a Turing machine M_a that computes $SHALT$, so that for all x,

$$\Phi(x,a) = SHALT(x) = \begin{cases} 1 & \text{if } M_x \text{ halts on } x \\ 0 & \text{otherwise.} \end{cases}$$

[6] More precisely: when M_n is started with $x+1$ 1s on the left end of the tape and blanks everywhere else, it eventually halts.

Then construct a Turing machine M_b[7] such that for all x,

$$\Phi(x,b) = \begin{cases} 7^{8} & \text{if } \Phi(x,a) = 0 \\ \uparrow & \text{if } \Phi(x,a) = 1 \end{cases} = \begin{cases} 7 & \text{if } SHALT(x) = 0 \\ \uparrow & \text{if } SHALT(x) = 1 \end{cases}$$

M_b operates as follows: it takes its input x and runs M_a on it to determine whether or not M_x halts on x. If M_a gives us a 0 verdict, i.e if M_x does not halt on x, then M_b outputs a 7 and halts; otherwise it gets itself into an infinite loop. Thus M_b halts on x iff M_x does not halt on x, or more formally, $(\forall x)[\Phi(x,b) \downarrow \iff \Phi(x,x) \uparrow]$ (I). From (I) and universal instantiation we obtain $\Phi(b,b) \downarrow \iff \Phi(b,b) \uparrow$, a contradiction. We thus conclude that the $SHALT$ function is not Turing–computable. ■

THEOREM 4.4.2. *The function*

$$HALT(x,y) = \begin{cases} 1 & \text{if } \Phi(x,y) \downarrow \\ 0 & \text{if } \Phi(x,y) \uparrow \end{cases} \text{ is not Turing–computable.}$$

Interpretation by Church's thesis: *There is no algorithm which, given an arbitrary Turing machine M and an arbitrary number x, can determine whether or not M halts on x.*

Proof. If there were a Turing machine that computed $HALT(x,y)$, we could use it to compute $SHALT(x)$ (simply let $y = x$). By the previous theorem, no Turing machine can compute $SHALT(x)$. Hence, by Modus Tollens, no Turing machine can compute $HALT(x,y)$. ■

THEOREM 4.4.3. *The function*

$$f_1(x) = \begin{cases} 1 & \text{if } \Phi(x,x) = 1 \\ 0 & \text{otherwise} \end{cases} \text{ is not Turing–computable.}$$

Proof. Suppose f_1 were Turing–computable. Then we would have a mechanical procedure which, given an arbitrary x_0, would decide whether or

[7] From now on we will often use phrases like "construct a machine M_b such that ...". What we mean is: construct a machine that behaves in the prescribed manner (as long as the prescription is algorithmic, Church's thesis will warantee that the construction is possible), and then, for the sake of the argument, assume that its godel number is b.

[8] There is nothing special about 7; any other number would do. As a convention, we will hereafter consistently use 7 whenever we need a dummy constant.

not M_{x_0} halts on x_0. Here it is: construct a Turing machine M_a such that for all x,

$$\Phi(x,a) = \begin{cases} 1 & \text{if } \Phi(x_0,x_0)\downarrow \\ \uparrow & \text{otherwise.} \end{cases}$$

Thus the behavior of M_a is totally irrelevant to its input x. No matter what input we give to it, M_a will invariably do the same thing: it will run M_{x_0} on x_0. If that computation ever converges then M_a will output an 1 and halt, otherwise it will get into an infinite loop. Now to find out whether $\Phi(x_0,x_0)\downarrow$, simply compute $f_1(a)$. If $f_1(a)=1$, then $\Phi(x_0,x_0)\downarrow$, while if $f_1(a)=0$, $\Phi(x_0,x_0)\uparrow$.

Thus if f_1 were Turing–computable, we would have an algorithm for computing the $SHALT$ function. But, by theorem 1 and Church's thesis, no such algorithm exists. Hence f_1 is not Turing–computable. ■

THEOREM 4.4.4. *The function*

$$f_2(x) = \begin{cases} 1 & \textit{if } \Phi_x \textit{ is total} \\ 0 & \textit{otherwise} \end{cases} = \begin{cases} 1 & \textit{if for all } y,\ \Phi(y,x)\downarrow \\ 0 & \textit{otherwise.} \end{cases}$$

is not Turing–computable.

Interpretation by Church's thesis: *There is no algorithm for determining whether or not a Turing machine computes a total function.*

Proof. Assume that f_2 is Turing–computable. Then here is an algorithm for computing the $SHALT$ function: given an arbitrary x_0, construct a Turing machine M_a such that for all x, $\Phi(x,a)=\Phi(x_0,x_0)$. In other words, regerdless of the value of its input x, all that M_a does is run M_{x_0} on x_0. Thus Φ_a is either (i) a constant function (namely, for all x, $\Phi_a(x)$ is the number $\Phi(x_0,x_0)$), or (ii) the nowhere-defined function (i.e for all $x, \Phi_a(x)\uparrow$), depending, respectively, on whether M_{x_0} halts on x_0 or not. To determine which is the case, we simply compute $f_2(a)$. If $f_2(a)=1$, (i) must be the case, so we must have $\Phi(x_0,x_0)\downarrow$. If $f_2(a)=0$, (ii) must be the case, so we must have $\Phi(x_0,x_0)\uparrow$. But, by theorem 1 and Church's thesis, there is no algorithm for computing the $SHALT$ function. Therefore, f_2 is not Turing–computable. ■

THEOREM 4.4.5. *The function*

$$f_3(x) = \begin{cases} 1 & \textit{if } W_x=\emptyset \\ 0 & \textit{otherwise} \end{cases} = \begin{cases} 1 & \textit{if } (\forall z)\Phi(z,x)\uparrow \\ 0 & \textit{otherwise} \end{cases} \textit{ is not Turing–computable.}$$

Interpretation by Church's thesis: *There is no algorithm for determining whether or not a Turing machine computes the nowhere–defined function.*

Proof. Assuming the contrary we derive an algorithm for $SHALT$ through the same construction we gave in the last theorem: to find out whether $\Phi(x_0, x_0) \downarrow$, for a given x_0, construct an M_a such that for all x, $\psi_{M_a}(x) = \Phi(x_0, x_0)$. Then if M_a computes the nowhere–defined function, $\Phi(x_0, x_0) \uparrow$, otherwise $\Phi(x_0, x_0) \downarrow$. Computing $f_3(a)$ would tell us what the case is. But the existence of such an algorithm is ruled out by theorem 1, and the result follows by contradiction. ■

THEOREM 4.4.6. *The function*

$$f_4(x) = \begin{cases} 1 & \textit{if } W_x \textit{ contains at least five numbers} \\ 0 & \textit{otherwise} \end{cases} \quad \textit{is not Turing–computable.}$$

Proof. If f_4 were computable, we could compute $SHALT$ by doing the following: given an x_0, construct an M_a such that for all x,

$$\Phi(x, a) = \begin{cases} 7 & \text{if } x < 4 \\ \Phi(x_0, x_0) & \text{if } x = 4 \\ \uparrow & \text{if } x > 4 \end{cases}$$

Then, to decide whether or not $\Phi(x_0, x_0)$, simply compute $f_4(a)$: if $f_4(a) = 1$, $\Phi(x_0, x_0) \downarrow$; if $f_4(a) = 0$, $\Phi(x_0, x_0) \uparrow$. By Church's thesis, this algorithm could be implemented by a Turing machine— which would thus compute the $SHALT$ function. No such Turing machine exists, hence f_4 is not Turing-computable. ■

THEOREM 4.4.7. *The function*

$$f_5(x) = \begin{cases} 1 & \textit{if } \Phi_x \textit{ is the successor function} \\ 0 & \textit{otherwise} \end{cases} = \begin{cases} 1 & \textit{if } (\forall z)[\Phi(z, x) = z + 1] \\ 0 & \textit{otherwise} \end{cases}$$

is not Turing–computable.

Proof. Once again, we reduce the halting problem to the computation of f_5. Given an x_0, construct an M_a such that for all x,

$$\Phi(x, a) = \begin{cases} x + 1 & \text{if } \Phi(x_0, x_0) \downarrow \\ \uparrow & \text{if } \Phi(x_0, x_0) \uparrow \end{cases}$$

M_a operates as follows: given an input x, it starts running M_{x_0} on x_0. If that computation ever converges, then M_a outputs the successor of x; otherwise it obviously diverges, i.e if $\Phi(x_0, x_0) \uparrow$ then Φ_a is the nowhere–defined function. Therefore, to determine whether M_{x_0} halts on x_0, we would simply compute $f_5(a)$ to determine whether M_a computes the successor function. But, by theorem 1 and Church's thesis, there is no algorithm for computing $SHALT$, hence F_5 cannot be Turing–computable. ■

COROLLARY 4.4.1. *The function*

$$f_6(x,y) = \begin{cases} 1 & \textit{if } \Phi_x = \Phi_y \\ 0 & \textit{otherwise} \end{cases} \quad \textit{is not Turing–computable.}$$

Interpretation by Church's thesis: *There is no algorithm which, given two arbitrary Turing machines, can determine whether or not they both compute the same function.*

Proof. Let M_a be a Turing machine that computes the successor function. Then, for all x, $f_6(x,a) = 1 \iff f_5(x) = 1$, so if we could compute f_6 we could also compute f_5. But f_5 is not Turing–computable, so the same must be the case for f_6. ■

THEOREM 4.4.8. *The function*

$$f_7(x,y) = \begin{cases} 1 & \textit{if } y \textit{ is in the range of } \Phi_x \\ 0 & \textit{otherwise} \end{cases} = \begin{cases} 1 & \textit{if } (\exists z)[\Phi(z,x) = y] \\ 0 & \textit{otherwise} \end{cases}$$

is not Turing–computable.

Proof. If we could compute f_7 we could also compute $SHALT$ as follows: given an x_0, construct an M_a such that for all x,

$$\Phi(x,a) = \begin{cases} 7 & \text{if } \Phi(x_0,x_0) \downarrow \\ \uparrow & \text{otherwise.} \end{cases}$$

Then to determine whether $\Phi(x_0, x_0) \downarrow$, simply compute $f_7(a, 7)$. If $f_7(a,7) = 1,\ \Phi(x_0,x_0) \downarrow$; if $f_7(a,7) = 0,\ \Phi(x_0,x_0) \uparrow$. But we know that $SHALT$ cannot be computed algorithmically, hence f_7 cannot be Turing–computable. ■

THEOREM 4.4.9. *The function*

$$f_8(x,y,z) = \begin{cases} 1 & \textit{if } \Phi(y,x) = z \\ 0 & \textit{otherwise} \end{cases} \quad \textit{is not Turing–computable.}$$

Proof. Construct an M_a exactly as we did in the last proof. Then for all numbers w we have $f_8(a, w, 7) = 1 \iff SHALT(x_0) = 1$. Therefore, an algorithm for f_8 would be tantamount to an algorithm for $SHALT$, and the result follows from the usual Modus Tollens argument. ■

THEOREM 4.4.10. *The function*

$$f_9(x) = \begin{cases} 1 & \text{if } W_x \text{ is finite} \\ 0 & \text{otherwise} \end{cases} \quad \text{is not Turing–computable.}$$

Interpretation by Church's thesis: *There is no algorithm for determining whether a Turing machine halts only for finitely many numbers.*
Proof. The assumption that f_9 is Turing–computable engenders the following algorithm for $SHALT$: given an x_0, construct a machine M_a such that for all x,

$$\Phi(x, a) = \begin{cases} 7 & \text{if } x = 0 \\ \Phi(x_0, x_0) & \text{if } x > 0 \end{cases}$$

Clearly, W_a is finite iff $\Phi(x_0, x_0) \uparrow$, so we could easily compute $SHALT(x_0)$ by computing $f_9(a)$. But no algorithm exists for $SHALT$, hence f_9 is not Turing–computable. ■

COROLLARY 4.4.2. *The function*

$$f_{10}(x) = \begin{cases} 1 & \text{if } W_x \text{ is infinite} \\ 0 & \text{otherwise} \end{cases} \quad \text{is not Turing–computable.}$$

Proof. From the previous theorem, since $f_{10}(x) = 1 \iff f_9(x) = 0$ ■.

THEOREM 4.4.11. *The function*

$$f_{11}(x) = \begin{cases} 1 & \text{if } \Phi_x \text{ is constant} \\ 0 & \text{otherwise} \end{cases} = \begin{cases} 1 & \text{if } (\exists y)(\forall z)\Phi(z, x) = y \\ 0 & \text{otherwise} \end{cases}$$

is not Turing–computable.
Interpretation by Church's thesis: *No algorithm can decide whether or not an arbitrary Turing machine computes a constant function.*
Proof. The same construction as in theorem 4: $f_{11}(a) = 1 \iff \Phi(x_0, x_0) \downarrow$. Church's thesis entails that f_{11} is uncomputable. ■

THEOREM 4.4.12. *The function*

$$f_{12}(x) = \begin{cases} 1 & \text{if } \overline{W_x} \text{ is finite} \\ 0 & \text{otherwise} \end{cases} \quad \text{is not Turing–computable.}$$

Interpretation by Church's thesis: *There is no algorithm which, given an arbitrary Turing machine M, can determine whether or not M diverges only on finitely many numbers.*

Proof. We reduce f_{12} to $SHALT$. Given an x_0, construct an M_a such that for all x,

$$\Phi(x,a) = \begin{cases} \uparrow & \text{if } x < 7 \\ \Phi(x_0,x_0) & \text{if } x \geq 7 \end{cases}$$

Clearly, $f_{12}(a) = 1 \Longleftrightarrow \Phi(x_0,x_0)\downarrow$. The usual line of reasoning finishes the proof. ■

Chapter 5

Computability with sets

Hereafter we will shift our attention from nt functions to sets of numbers, i.e to the power–set of N. This is not an essential change since functions are themselves sets. Of course they are not sets of numbers, they are sets of *pairs* of numbers, or, more generally, finite sequences of numbers. But we have already seen that finite sequences of numbers can be uniquely encoded by single numbers. Thus we can uniquely represent a function $f : N^k \longrightarrow N$ by the set $\{ [x_1, \ldots, x_k, y] \mid x_1, \ldots, x_k, y \in N \text{ and } y = f(x_1, \ldots, x_k)\}$.

We say that a set A is recursive iff its characteristic function

$$c_A(x) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{if } x \notin A \end{cases}$$

is Turing–computable. Informally, then, a set A is recursive iff there is an algorithm which can determine, for arbitrary x, whether or not $x \in A$. Examples of recursive sets: $N = \{0, 1, 2, 3, \ldots\}$ (the computation of $c_N(x)$ is very simple: we invariably output an 1), the empty set $\emptyset$ (invariably output a zero), the odd numbers (output $x \, mod \, 2$), the prime numbers, the set $\{x \in N \mid P_x \text{ has } 2x \text{ instructions}\}$, etc. The following is an immediate consequence of our definition:

THEOREM 5.1. *A set A is recursive iff its complement $\overline{A}$ is recursive.*

Another simple but fundamental result is the following:

THEOREM 5.2 *All finite sets are recursive.*

Proof. Essentially we have already proven this when we discussed the computability of finite functions (section 3.1). To recapitulate, here is a Pascal subroutine that will compute the characteristic function of any finite set $A = \{a_1, \ldots, a_n\}$:

```
FUNCTION C_A (x : INTEGER): INTEGER;
```

```
BEGIN
    IF x = a_1 THEN C_A := 1
    ELSE
    IF x = a_2 THEN C_A := 1
    ELSE
    ⋮
    IF x = a_n THEN C_A := 1
    ELSE
       C_A := 0;
END;
```

By theorem X, c_A is Turing–computable. ■

A set A is called **recursively enumerable** (abbreviated **re**) iff there is an algorithm that will enumerate all and only the members of A in a list $a_0, a_1, a_2, \ldots$. Of course if A is infinite the algorithm will keep going *ad infinitum*. We allow for the possibility that some elements of A might appear in the list more than once. All we require is that *every* element of A gets listed at least once and that *only* elements of A get listed. In addition, it is convenient to regard the empty set $\emptyset$ as re.

We can define re sets more formally by invoking Church's thesis:
A set A is re iff (i) $A = \emptyset$, or (ii) there is a Turing machine M such that the list $\psi_M(0), \psi_M(1), \psi_M(2), \ldots$ comprises all and only the members of A, i.e such that $A = \{\psi_M(x) \mid x \in N\}$.[1] We can prove that

THEOREM 5.3. *All recursive sets are re.*

Proof. Let $A \neq \emptyset$ be any recursive set (if $A = \emptyset$ it is re by definition). Let $a_0 = min_x[c_A(x) = 1]$. a_0 is the smallest element of A (one must exist since A is non–empty) and can be computed mechanically since c_A is Turing–computable. Define the function

$$f(x) = \begin{cases} x & \text{if } c_A(x) = 1 \\ a_0 & \text{if } c_A(x) = 0 \end{cases}$$

Now f is obviously mechanically computable, so, by Church's thesis, it is also Turing–computable. Furthermore, it is total and its range is A (i.e $\{f(x) \mid x \in N\} = A$), thus A is re. ■

An alternative definition of recursive enumerability is the following:
A set A is re iff it is the domain of a Turing–computable function, i.e iff

[1] Condition (ii) is tantamount to the existence of a Turing–computable function f which is total and whose range is A, i.e such that $A = \{f(x) \mid x \in N\}$.

there is a Turing machine M_a such that $A = W_a = \{x \in N \mid \Phi_a(x)\downarrow\}$. The proof that this is equivalent to our first definition is a good opportunity to introduce the important technique of dovetailing:

THEOREM 5.4. *The two definitions are equivalent.*

Proof. Suppose that a set A satisfies the first definition. There are two cases:

(i) $A = \emptyset$**.** Then $A = W_a$ for any machine M_a that computes the nowhere-defined function.

(ii) There is a machine M_a such that Φ_a is total (i.e M_a halts for all $x \in N$) and $A = \{\Phi_a(x) \mid x \in N\}$. In that case construct a Turing machine M_b such that for all x, $\Phi(x, b) = min_y[\Phi_a(y) = x]$. In other words, given its input x, M_b runs M_a on 0 (this must terminate since Φ_a is total) and then checks whether the output $\Phi_a(0)$ is equal to x. If it is, it halts, otherwise it proceeds to check whether $\Phi_a(1) = x$. If that doesn't hold either, it goes on with $\Phi_a(2)$, then $\Phi_a(3)$, and so forth. Clearly, the process will stop iff $\Phi_a(y) = x$ for *some* y, i.e iff $x \in A$. Therefore, $(\forall x)[M_b$ halts on $x \iff x \in A]$, in accordance with the second definition.

Now in the other direction, suppose that there is a Turing machine M_a which halts for all and only the elements of A, i.e $(\forall x)[\Phi_a(x)\downarrow \iff x \in A]$. Now A is either finite or infinite. If it is finite then it must also be recursive (theorem 5.2), and by the proof of theorem 5.3, it must also be re in the sense of the first definition. Assuming then that A is infinite, we present an algorithm for constructing a list $L: a_0, a_1, a_2, \ldots$ which comprises all and only the elements of A. The construction proceeds in stages $s = 1, 2, 3, \ldots$. Initially we let L be the empty list.

Stage 1:

- Compute 1 step in the computation $\Phi_a(0)$. If M_a halts on 0 in that first step, i.e if $STEP(0, a, 1) = 1$, append 0 to L. Otherwise proceed to the next stage.

Stage 2:

- Compute 2 steps in the computation $\Phi_a(0)$. If $STEP(0, a, 2) = 1$, append 0 to L.
- Compute 2 steps in the computation $\Phi_a(1)$. If $STEP(1, a, 2) = 1$, append 1 to L.

Stage 3:

- Compute $STEP(0, a, 3)$. If the result is 1, append 0 to L.

- Compute $STEP(1, a, 3)$. If the result is 1, append 1 to L.
- Compute $STEP(2, a, 3)$. If the result is affirmative, append 2 to L.

$\vdots$

Stage s:

Compute $STEP(0, a, s), STEP(1, a, s), \cdots, STEP(s-1, a, s)$, in that order. If $STEP(i, a, s) = 1$ (as we go through $i = 0, 1, 2, \ldots, s-1$), append i to L.

The object of this algorithm is essentially to find out whether $\Phi_a(i) \downarrow$ for $i = 0, 1, 2, \ldots$. But instead of attempting to compute $\Phi_a(i)$ all at once and run the risk of getting stuck in an infinite loop (in case $\Phi_a(i) \uparrow$), it progressively interleaves the computations $\Phi_a(0), \Phi_a(1), \ldots, \Phi_a(i)$ to greater and greater "depths" (numbers of steps), and greater and greater i. This rather clever and very useful method is known as **dovetailing** (in honor of its originator Wood U. Dovetail). Clearly, if M_a halts on *any* number i and in *any* number of steps s, our dovetailing procedure will eventually discover it. Therefore,
(I) A number i will be appended to L iff $\Phi_a(i) \downarrow$, i.e iff $i \in A$. [2]
And since A is infinite, (I) implies that
(II) L will comprise infinitely many numbers.

Now define the function $f(x) = a_x =$ the x^{th} member of L. (II) implies that f is total, while (I) implies that $A = \{f(x) \mid x \in N\}$. And since we have given a rigorous algorithm for computing $f(x)$, Church's thesis implies that f is Turing–computable. Thus A is re in accordance with the first definition. ■

Therefore, the list $W_0, W_1, W_2, \ldots$ is an enumeration of all and only the re sets.

A fundamental relation between recursiveness and recursive enumerability is the following:

THEOREM 5.5. *A set A is recursive iff both A and $\overline{A}$ are re.*

Proof. The "only if" part follows directly from theorems 5.1 and 5.3. For the converse, let M_a and M_b be two Turing machines that halt for all and only the elements of A and $\overline{A}$, respectively.[3] Then here is an algorithm for computing $c_A(x)$: start computing both $\Phi_a(x)$ and $\Phi_b(x)$ simultaneously. Since either $x \in A$ or $x \in \overline{A}$, one and only one of these computations must

[2] In fact every $i \in A$ will be appended to the list infinitely many times, but as we have already said, repetitions do not bother us.

[3] We are using the second definition here.

eventually halt. If $\Phi_a(x)$ halts let $c_a(x) = 1$ and if $\Phi_b(x)$ halts let $c_A(x) = 0$. By Church's thesis, c_A is Turing–computable and A is recursive. ■

Next we define a few sets that will play a rather important role in the last part:

$$K = \{x \in N \mid \Phi(x,x)\downarrow\}$$

$$KH = \{x \in N \mid \Phi(l(x), r(x))\downarrow\}$$

$$TOT = \{x \in N \mid \Phi_x \text{ is total}\} = \{x \in N \mid (\forall z)\ \Phi_x(z)\downarrow\}$$

$$UNDEF = \{x \in N \mid W_x = \emptyset\} = \{x \in N \mid (\forall z)\ \Phi_x(z)\uparrow\}$$

$$FIN = \{x \in N \mid W_x \text{ is finite}\}$$

$$INF = \overline{FIN} = \{x \in N \mid W_x \text{ is infinite}\}$$

$$CONST = \{x \in N \mid \Phi_x \text{ is constant}\} = \{x \in N \mid (\exists c)(\forall z)\ \Phi_x(z) = c\}$$

$$COFIN = \{x \in N \mid W_x \text{ is co-finite}\} = \{x \in N \mid \overline{W_x} \text{ is finite}\}$$

THEOREM 5.6. *None of the above sets is recursive.*
Proof. By theorems 4.4.1, 4.4.2, 4.4.4, 4.4.5, 4.4.10, corollary 4.4.2, and theorems 4.4.11 and 4.4.12. ■

Note that although K is not recursive, it is certainly re, as one can easily construct a Turing machine M_k such that for all x, $\Phi(x,k) = \Phi(x,x)$. Thus M_k will halt on input x iff $\Phi(x,x)\downarrow$, so that $K = W_k = \{x \mid \Phi(x,x)\downarrow\}$. Now since K is re and non–recursive, the contrapositive of theorem 5.5 implies
THEOREM 5.7. *The set* $\overline{K} = \{x \in N \mid \Phi(x,x)\uparrow\}$ *is not re.*

Part II

Relative computability

Chapter 6

Reducibilities

Our formalizations of problems as sets of natural numbers and of mechanical procedures as Turing machines enabled us to divide the "class of all problems" (fig. 6.1) into two basic sub-classes: the class of solvable problems[1] (region A) and the class of unsolvable problems (region B). This fundamental dichotomy becomes the departing point for two diverging branches of Theoretical Computer Science: complexity theory and unsolvability theory. Complexity theory focuses on region A. It aims at classifying solvable problems on the basis of the time and space resources that are needed for their solution. Such classifications subdivide region A into several classes of problems (polynomial, exponential, etc.), which are then further partitioned in accordance with even finer structural distinctions, and so on.

The other branch, the "theory of unsolvable problems", or "relative computability", concentrates on region B. It is this theory that we take up in this last part of the essay. Most of the motivation for studying relative computability stems from asking "What if ...?" questions. In particular, all of the unsolvable problems that were presented in Part I were proven unsolvable by showing that *if* we could compute any one of them, *then* we would be able to compute K, the halting problem; and then of course we went on to say that K was proven to be non-recursive, therefore .... So one question that naturally arises is: how about the other way? *If* we could solve the halting problem, would we then be able to solve all of those other problems as well? Would there remain any unsolvable problems at all if K were recursive? Or would we then be able to compute everything? If yes, then K would turn out to be the touchstone of unsolvability, the apex

[1] For brevity's sake we will hereafter simply say 'solvable'/'unsolvable' instead of 'mechanically solvable'/'mechanically unsolvable'.

Figure 6.1: The class of all problems

of the pyramid of all problems. If not, then that would mean that there are problems which are "more unsolvable" than the halting problem, in the sense that *even if we could compute* K, we would still be unable to solve those other problems.

More generally, we ask: are all unsolvable problems equally unsolvable (so that if we could compute one of them we could compute all of them), or are there finer shades of unsolvability to be distinguished? We will make this fundamental question the starting point of our inquiry. We will soon discover that there are, indeed, different degrees of unsolvability. In fact we will see that there are infinitely many such degrees, and we will get to examine their mathematical structure in considerable detail. We will also see, especially in connection with Posts's problem, that there is a great deal of interplay between relative computability and the subject of first-order theories; the study of the so-called "recursively enumerable degrees", in particular, will be seen to afford us many valuable semantic insights into the nature of first-order mathematical systems. But before we plunge ahead into the technical details, it will be a good idea to illustrate the basic concepts of relative computability from an informal, intuitive perspective. We do this in the next section by way of a mathematical parable.

6.1 An informal explanation of reducibility and degrees of unsolvability.

Our thought experiment will be based on the difficult problems which seem to be perpetually besetting David Problemfrought, a senior high school student in a small NorthEastern town. Let us use the term 'david-problem' to refer to any problem[2], be it mathematical or empirical, that may be meanigfully posed to our friend David, and let us say that a d-problem is d-solvable (short for 'david-solvable') iff David can provide a satisfactory solution to it. An obvious defect of these definitions is the inability to tell with certainty whether or not a given problem is d-solvable: a problem for which David has no solution at some given time might become trivial for him at some later time. For example, he might not know Calculus and thus be unable to differentiate a polynomial; later on in time, however, he might very well

[2]Here we will be using the term 'problem' in a rather broad sense, including what we (in chapter 2) called "concrete questions".

learn all about Calculus and differentiation. Now are we to say that the problem of differentiating $3x^2 + 5x$ is d-solvable or not? Likewise, a problem which can be confidently pronounced d-solvable at a certain time t_1 might very well become d-unsolvable at some later time t_2. Now it is clear that the source of this problem is temporal ambiguity, so the obvious solution is to "freeze time", i.e to select an arbitrary time slice (of reasonable brevity) and then define the concepts in question with respect to that period. If we do this, i.e if we choose something like May 18, 1994 as our central point of reference, and if we can tolerate a meager dose of ambiguity on what constitutes a d-problem or a d-solution, then it seems fairly sensible to talk about the class of all d-problems with the understanding that each such problem is either d-solvable or not. Having made these conventions, I will now plow ahead without any further ado.

The CAR problem, a david-problem of considerable urgency, arises in the following scenario. It is a beautiful Spring Friday (May the 18^{th}, of course) and David's parents have left for the weekend. They took the car of David's mother, which means that the car of David's father is sitting all alone in the garage, unused and alluringly tempting. Even though his father did not give him the necessary permission, David feels a severe need to take the car out for a ride, and would definitely do so were it not for the CAR problem: there are only two keys to the car, one of which is away with his father, while the other is inside the first drawer of the living-room dresser. David decides to pay a visit to the dresser and for a moment he is convinced that triumph is imminent, but alas, he now runs into the $DRAWER$ problem: the drawer itself is locked and, unlike the car, this drawer has only one key, which, of course, is also in the possession of David's father. Now a solution to the CAR problem would be to obtain one of the two car keys; then David could unlock the car and drive away in pursuit of all the excitement that befitts an adventuresome youth like himself. Since, however, this is in fact impossible, we can say that for all practical purposes the CAR problem is d-unsolvable. The same goes for the $DRAWER$ problem: since the only key to the drawer is in the possession of David's father, who is away, we may reasonably say that the $DRAWER$ problem is also d-unsolvable.

Since both problems are d-unsolvable, a practical-minded theory of d-solvability would take no further interest in them. Such a theory would restrict its attention to the d-solvable problems, i.e problems which we know David can solve, with a view to classifying them according to the complexity of their solutions—perhaps depending on whether David can solve them in polynomial as opposed to exponential time, etc. On the other hand, a more abstract-minded theory of d-unsolvability will continue the study of

the CAR and $DRAWER$ problems even after their unsolvability has been established, in an attempt to discover exactly *how* unsolvable they are, and thus classify them according to the **degree** of their unsolvability.

Now at first glance this aim might appear counter-intuitive, or even nonsensical. We are inclined to think that there cannot be different "degrees" of unsolvability; any given problem is either d-solvable or d-unsolvable and that's all there is to it. This inclination might not be misguided if we are looking at two problems *individually*, but the point is that if we do that we will be overlooking the possibility that some subtle reducibility relationship might in fact exist *between them*. In particular, we will be overlooking the possibility that one of the problems might be d-solvable *relative to the other*—even though both of them may well be d-unsolvable in actuality. To make this idea a little more precise, let us say that *for any two d-problems p_1 and p_2, p_1 is* **d-reducible** *to p_2, written as $p_1 \leq_d p_2$, iff a solution to p_2 will automatically enable David to solve p_1.*

Then it is not difficult to see that $CAR \leq_d DRAWER$, because opening the drawer is essentially tantamount to opening the car. Now if we futhermore agree to write $p_1 \not\leq_d p_2$ iff it is not the case that $p_1 \leq_d p_2$, then it is equally easy to see that $DRAWER \not\leq_d CAR$, because a hypothetical solution to the CAR problem does *not* provide a solution to the $DRAWER$ problem: even if David somehow manages to get into the car, he will still be unable to open up the drawer. Therefore, it seems intuitively sound, albeit a little odd, to say that the $DRAWER$ problem is "more difficult" or "more unsolvable" than the CAR problem; or, equivalently, that the CAR problem is "easier", or "less unsolvable" than the $DRAWER$ problem. For even though both problems are in fact d-unsolvable, they are not quite of the same degree of unsolvability: a hypothetical solution to one of them immediately engenders a solution to the other, but the converse is not true. And this serves to distinguish them from a relative point of view and put them at different levels of d-unsolvability.

The question now suggests itself, can there be two d-unsolvable problems of the same level of difficulty? Of course there can, as the following possible scenario demonstrates: suppose that David's father recently made a second copy of the drawer key, which he decided to keep permanently inside the glove compartment of his car, and suppose moreover that David is aware of that. Then in this modified situation we not only have $CAR \leq_d DRAWER$, but also $DRAWER \leq_d CAR$, because if David could now open the car, he would also be able to get a hold of the second copy of the drawer key and thus solve the $DRAWER$ problem. Let us denote a case like this, in which two d-problems p_1 and p_2 are reducible to each other, i.e a case in which

$p_1 \leq_d p_2$ *and* $p_2 \leq_d p_1$, by writing $p_1 \equiv_d p2$; and let us furthermore say of two such problems that they are of "the same degree of d-unsolvability". Then in the context of our new scenario and by virtue of our new terminology, we have $CAR \equiv_d DRAWER$, i.e these two problems are of the same degree of d-unsolvability. This result certainly coheres with our intuition, which tells us that CAR and $DRAWER$ are now of the same level of difficulty, or equally unsolvable, even from a relative point of view.

Let us now see how our definition of david-reducibilty applies to boundary cases, i.e cases in which we have two d-solvable problems, or one d-solvable and one d-unsolvable problem. The following result applies to such cases:

PROPOSITION 6.1.1. *If p_1 and p_2 are two d-solvable problems and p_3 is a d-unsolvable problem, then we have*
(1) $p_1 \equiv_d p_2$
(2) $p_1 \leq_d p_3,\ \ p_2 \leq_d p_3$
(3) $p_3 \not\leq_d p_1,\ \ p_3 \not\leq_d p_2$.

These results are also consistent with our intuition. Recall that we have defined a problem p to be d-reducible to a problem q ($p \leq_d q$) iff a solution to q will automatically enable David to solve p. But if David can already solve p, i.e if p is d-solvable, then we will (vacuously) have $p \leq_d q$ regardless of whether or not q is d-solvable. That is the content of assertions (1) and (2). And (3) obviously holds because no unsolvable problem can be reduced to a solvable one; no matter how many solvable problems David solves, he will still be unable to enter the CAR, as that problem is in fact d-unsolvable.

Are any two d-unsolvable problems somehow related to each other via the $\leq_d$ relation? Of course not. For it is quite possible that the two problems should be totally unrelated, in which case the solution of either one would have no bearing on the other. Consider for example the $PROM\_DATE$ problem: it is Friday, May 18, and the prom dance of David's senior class is happening tomorrow, Saturday evening at 8:00 pm. David has been trying to solve this problem for quite a while now, but to no avail. Since there are fewer than 24 hours left until the party, we may say that for all practical purposes the $PROM\_DATE$ problem is d-unsolvable. It is obvious, however, that there is no connection between this problem and the CAR or the $DRAWER$ problems. The only thing that all three have in common is that they are d-unsolvable. Other than that, we have, for instance, neither $CAR \leq_d PROM\_DATE$ nor $PROM\_DATE \leq_d CAR$. Any two such problems p_1 and p_2 for which $p_1 \not\leq_d p_2$ and $p_2 \not\leq_d p_1$ will be called **incomparable**. On the other hand, the $JENNY$ problem, the problem of conquering his classmate's Jenny's heart (towards the solution of which David has expended

sizeable amounts of energy) is certainly comparable to the $PROM\_DATE$ problem. In particular, we have $PROM\_DATE \leq_d JENNY$, though it is easy to see that $JENNY \not\leq_d PROM\_DATE$, which means that, as David is already painfully aware, the $JENNY$ problem is of a higher degree of unsolvability than the $PROM\_DATE$ problem.

Observe that the $\equiv_d$ relation, being reflexive, symmetric, and transitive, is an equivalence relation. It thus partitions the class of all d-problems into disjoint classes, each class comprising problems that are $\equiv_d$-equivalent, i.e mutually reducible to one another, or "equally unsolvable". These classes are what we will mean by 'degrees of unsolvability'. More precisely, we define $deg(p)$, the **degree of unsolvability of a d - problem p**, to be its $\equiv_d$-equivalence class, i.e $deg(p) = \{q \mid p \equiv_d q\}$. Hence p's degree of unsolvability *is* the class of all and only those problems that are exactly as unsolvable as p.

Note that (1) there is *only one* degree that contains d-solvable problems, and that (2) that one degree contains all and only the d-solvable problems. (1) is true because all solvable problems are trivially reducible to one another, and (2) is true because no unsolvable problem can be reduced to a solvable one. Some other immediate consequences of the definition of degrees are given by the following:

PROPOSITION 6.1.2. *For all d-problems p, p_1, p_2, and degrees d_1 and d_2, we have:*
(i) $p \in deg(p)$,
(ii) $p_1 \equiv_d p_2 \iff deg(p_1) = deg(p_2)$,
(iii) $d_1 \neq d_2 \implies d_1 \cap d_2 = \emptyset$.

All of the above follow by virtue of the fact that degrees are equivalence classes. (ii) simply reiterates that p_1 and p_2 are reducible to each other iff they belong to (or "are of") the same degree of unsolvability; (iii) says that no problem can be of two different degrees of unsolvability.

We finally induce a partial order on degrees that will ultimately arrange the class of all d-problems in the manner depicted in fig. 6.2. In particular, we write $d_1 \preceq_d d_2$ and say that the degree d_1 is **lower than** or **below** d_2, iff $p_1 \leq_d p_2$ for some problems $p_1 \in d_1$ and $p_2 \in d_2$[3]; we will write $d_1 \not\preceq_d d_2$ if this is not the case. Consequently we see that $deg(CAR) \preceq_d deg(DRAWER)$ while $deg(DRAWER) \not\preceq_d deg(CAR)$ (in our first, unmodified scenario). Similarly,

$$deg(PROM\_DATE) \preceq_d deg(JENNY),$$

[3]Or, equivalently, for *all* problems $p_1 \in d_1$ and $p_2 \in d_2$, because if $p_1 \leq_d p_2$ for *some* $p_1 \in d_1$, $p_2 \in d_2$, then $p \leq_d q$ for *all* $p \in d_1$, $q \in d_2$—and conversely.

while

$$deg(JENNY) \not\preceq_d deg(PROM\_DATE).$$

Also, if *dsol* is the degree of all d-solvable problems, then for any degree d we have $dsol \preceq_d d$. Finally, if for two degrees d_1 and d_2 we have neither $d_1 \preceq_d d_2$ nor $d_2 \preceq_d d_1$, we will say that d_1 and d_2 are incomparable. $Deg(JENNY)$ and $deg(DRAWER)$ are two such degrees.

It is easily verified that $\preceq_d$ is a partial order, i.e reflexive, anti-symmetric and transitive. Fig. 6.2 is a sketchy display of the structure imposed on the class of all d-problems by this partial order.

6.2 Many-one reducibility

We will now seek to formalize the intuitive notions that we developed in the last section. We will begin with the reducibility relation (which is now to hold between two sets of numbers), since everything else can be subsequently defined in terms of it.

Our strategy will be to analyze the reductions that were carried out in section 4.4 with a view to abstracting from them a formal definition. To that end, consider, for instance, the proof of unsolvability of $FIN = \{x \in N \mid W_x \text{ is finite }\}$. The proof shows that K is reducible to FIN by exhibiting an algorithm which, given an arbitrary number x_0 produces a number a such that $x_0 \in K \iff a \in FIN$. The algorithm is simple: given the x_0, we construct a Turing machine M_a such that for all x, $\Phi(x, a) = \Phi(x_0, x_0)$. Evidently, $x_0 \in K \iff a \in FIN$, and thus the computation of K is reduced to the computation of FIN.

Dissection of the other proofs reveals that there is indeed a common pattern to all of them: they show that K is reducible to the problem at hand, say P, by establishing the existence of an algorithm, which, given an arbitrary number x_0, produces a number a such that $x_0 \in K \iff a \in P$. We now generalize this observation and say that *a problem P is reducible to a problem Q iff there is an algorithm which, given any number x, produces a number a such that* $x \in P \iff a \in Q$. The algorithm, of course, must halt and produce an appropriate output a for *any* input x. We advance to a higher degree of formalism by employing Church's thesis and replacing 'algorithm' with 'Turing machine': *A problem P is reducible to a problem Q iff there is a Turing machine M which halts for all inputs x and is such that* $x \in P \iff \psi_M(x) \in Q$. We arrive at our final definition of reducibility (which we call 'm-reducibility' for reasons that will be explained shortly) by replacing the 'Turing machine M which halts for all inputs x' by the 'total

Figure 6.2: A partial ordering of the d-unsolvability degrees.

function f' that M computes, and 'problem' by its extensional equivalent 'set':

DEFINITION 6.1.1. *A set A is* **many-one reducible** *to a set B, abbreviated as 'm-reducible' and written as $A \leq_m B$, iff there is a total Turing-computable function f such that for all x, $x \in A \iff f(x) \in B$. We write $A \not\leq_m B$ if no such function exists.*

We call this reducibility 'many-one' because it allows for the possibility that f should have the same value for different arguments. In other words, it does not require that f should be one-one. If we impose this additional requirement we wind up with a different reducibility relation (suitably called 'one-one reducibility'), which will be discussed in the next section. For now we focus on many-one reducibilty, and we begin by proving some of its fundamental prorerties:

THEOREM 6.1.1. *(i) If A and B are two recursive sets other than N and $\emptyset$, then $A \leq_m B$ and $B \leq_m A$.*
(ii) $\leq_m$ is reflexive and transitive.
(iii) For all sets A and B, $A \leq_m B \Longrightarrow \overline{A} \leq_m \overline{B}$.
(iv) If B is recursive and $A \leq_m B$, then A is recursive.
(v) If B is re and $A \leq_m B$, then A is re.

Proof. (i) To show that $A \leq_m B$: let b and b' be any two numbers in B and $\overline{B}$, respectively (such numbers must exist since both B and $\overline{B}$ are non-empty). Then define the function

$$f(x) = \begin{cases} b & \text{if } x \in A \\ b' & \text{if } x \notin A. \end{cases}$$

Now $x \in A$ and $x \notin A$ are both mechanically computable predicates (since A is recursive), so f is mechanically and thus Turing-computable. In addition, it is total and such that $(\forall x)[x \in A \iff f(x) \in B]$. Thus we conclude that $A \leq_m B$. A perfectly symmetrical argument shows that $B \leq_m A$.

(ii) Reflexivity is established through the obviously computable identity function $i(x) = x$. For transitivity: assuming that $A \leq_m B$ and $B \leq_m C$, so that for all x, $[x \in A \iff f(x) \in B]$ and $[x \in B \iff g(x) \in C]$ (for two appropriate functions f and g), we get $A \leq_m C$ via the composition $h(x) = g(f(x))$.

(iii) The same fuction which proves that $A \leq_m B$ proves that $\overline{A} \leq_m \overline{B}$.

(iv) Let $A \leq_m B$, where B is recursive. Then there is a total Turing-computable function f such that $(\forall x)[x \in A \iff f(x) \in B]$ (I). So, to determine whether $x \in A$, for a given x, simply compute $f(x)$ and check whether it is in B (which we can do mechanically, since B is recursive). If

it is, then, by (I), $x \in A$; if not, then $x \notin A$.
(v) Same idea: let $x \in A \iff f(x) \in B$, for some re B. By definition of re, there is a Turing machine M_b such that $W_b = B$. Construct a machine M_a such that for all x, $\Phi_a(x) = \Phi_b(f(x))$. Clearly, [M_a halts on x] $\iff$ [M_b halts on $f(x)$] $\iff f(x) \in B \iff x \in A$, and thus A is re. ■

Reducibility having been defined, the other concepts follow easily:
(i) *Two problems A and B are* **equally m-unsolvable** *iff $A \leq_m B$ and $B \leq_m A$; we denote this by writing $A \equiv_m B$.*
(ii) *A problem B is* **more m-unsolvable** *than a problem A (or A is 'less m-unsolvable' than B) iff $A \leq_m B$ but $B \not\leq_m A$.*

These definitions naturally pave the way for the definition of m-degrees of unsolvability. We take the same approach we took for david-reducibility: we define the m-degree of unsolvability of a problem A to be the class of all and only those problems B which are equally m-unsolvable with A. Of course this class is nothing but A's equivalence class with respect to the $\equiv_m$ relation. Formally, we say that the **m-degree**[4] **of unsolvability** of A is the class $deg_m(A) = \{B \subseteq N | A \equiv_m B\}$. As with david-degrees, for any two problems A and B we have
(1) $A \in deg_m(A)$,
(2) $deg_m(A) = deg_m(B) \iff A \equiv_m B$, and
(3) if $deg_m(A) \neq deg_m(B)$ then $deg_m(A) \cap deg_m(B) = \emptyset$.
We will use lower-case boldface letters like $\mathbf{a}, \mathbf{a'}, \mathbf{b}, \mathbf{b_1}, \ldots$, to denote m-degrees.

Next we want to order m-degrees so that we can speak of such a degree $\mathbf{a}$ as being "lower" or "higher" than another degree $\mathbf{b}$, etc. We define a binary relation $\mathbf{a} \preceq_m \mathbf{b}$ to hold iff the problems in $\mathbf{a}$ are m-reducible to those in $\mathbf{b}$, i.e iff $A \leq_m B$ for some (any) sets $A \in \mathbf{a}$ and $B \in \mathbf{b}$. This relation is reflexive, anti-symmetric, and transitive, and thus it induces a partial order. Now, intuitively, an m-degree $\mathbf{a}$ should be said to be "lower" than $\mathbf{b}$ iff the problems in $\mathbf{a}$ are m-reducible to those in $\mathbf{b}$ *but not conversely*, i.e iff $\mathbf{a} \preceq_m \mathbf{b}$ *and* $\mathbf{b} \not\preceq_m \mathbf{a}$. Since, by the anti-symmetry of $\preceq_m$, this conjunction is true iff $\mathbf{a} \preceq_m \mathbf{b}$ and $\mathbf{a} \neq \mathbf{b}$, we arrive at the following definition: we say that *an m-degree $\mathbf{a}$ is lower than an m-degree $\mathbf{b}$ (or, equivalently, $\mathbf{b}$ is higher than $\mathbf{a}$) iff $\mathbf{a} \preceq_m \mathbf{b}$ and $\mathbf{a} \neq \mathbf{b}$; we denote this by writing $\mathbf{a} \prec_m \mathbf{b}$, or, equivalently, $\mathbf{b} \succ_m \mathbf{a}$.*

We are now ready to examine the structure of the $\preceq_m$ ordering in detail. Let us begin with the solvable region, i.e the recursive sets. From our expe-

[4] We call these 'm-degrees' to emphasize that they arise from the many-one reducibility relation.

rience with david-reducibility, we should expect all the solvable problems to be m-reducible to one another and thus belong to one single degree, which should be the lowest of all m-degrees. This is almost true, but not entirely so on account of two pathological cases: the set of all numbers N and the empty set $\emptyset$. Except from these two, any other recursive sets A and B are m-reducible to each other (th. 6.2.1.i) and therefore belong to the same m-degree, let us call it REC. In other words, REC contains all and only the recursive sets besides N and $\emptyset$. Now why cannot N be in REC? Because if it were, we should have $A \equiv_m N$ for any recursive set A, i.e there should be a total Turing-computable function f such that $(\forall x)\ (x \in A \iff f(x) \in N)$, or, equivalently, $x \notin A \iff f(x) \notin N$; clearly, no such function exists. In fact N cannot co-exist in the same m-degree with *any* other set, recursive or not, precisely for that reason. Consequently, it constitutes a degree all on its own, meaning that $deg_m(N)$ has only one member, N itself. The same remarks apply to the empty set $\emptyset$, which makes up another degree by itself. These two degrees are clearly incomparable, since there is no total function f such that $x \in N \iff f(x) \in \emptyset$ or $x \in \emptyset \iff f(x) \in N$. Both, however, are lower than REC, i.e (a) $deg_m(N) \prec_m REC$, and (b) $deg_m(\emptyset) \prec_m REC$. To see (a), note that for any recursive set A we have $N \leq_m A$, i.e $(\forall x)\ [x \in N \iff f(x) \in A]$, by letting f be the total Turing-computable function that enumerates A.[5] For (b), note that for any recursive set A we have $(\forall x)\ [x \in \emptyset \iff g(x) \in \overline{A}\ ]$.[6] Finally, it should be evident that $deg_m(N)$ and $deg_m(\emptyset)$ are the lowest m-degrees. Graphically, the conclusions we have drawn so far are depicted in fig. 6.3.

We now come to the questions that concern us most:
(1) Are all unsolvable problems equally m-unsolvable, or are there different m-degrees of unsolvability?
(2) Are all unsolvable problems $\leq_m$-comparable?
Both questions have negative answers, as the following two theorems show:
THEOREM 6.2.2. *There is a problem A that is more m-unsolvable than the halting problem, i.e such that $K \leq_m A$ but $A \not\leq_m K$. Consequently, there are at least two different non-recursive m-degrees of unsolvability.*
Proof. Take $A = TOT = \{x \in N \mid \Phi_x \text{ is total } \} = \{x \in N \mid\ (\forall z)\ \Phi_x(z) \downarrow \}$. We proved in ch. 4 that $K \leq_m TOT$, which is how we concluded that TOT is not recursive. We must now show that $TOT \not\leq_m K$. Since K is re, it would follow from theorem 6.2.1.iv that *if* we had $TOT \leq_m K$, then TOT

[5] A is recursive, so it must also be re, i.e there must be a total Turing-computable function f such that $A = \{f(x) \mid x \in N\}$.

[6] Likewise, since A is recursive, both it and $\overline{A}$ are re, so $\overline{A}$ must also be the range of some total Turing-computable function g.

Figure 6.3: Partition of the power-set of N by $\equiv_m$.

would also be re. Hence, if we show that TOT is *not* re, we will have shown that $TOT \not\leq_m K$. This is what we will do, by way of contradiction. In particular, assume that TOT is re. By definition of re, there must be a total Turing-computable function f which enumerates TOT, i.e such that $TOT = \{f(x) \mid x \in N\}$. Now construct a Turing machine M_a such that for all $x \in N$, $\Phi(x, a) = \Phi(x, f(x)) + 1$. The machine operates as follows: given an input x, it runs $M_{f(x)}$ on x, it obtains a result (this is guaranteed because $f(x) \in TOT$, so we know that $M_{f(x)}$ will halt on x), and then it adds one to it. Since this procedure must terminate for all x, we conclude that $(\forall x)\ \Phi_a(x)\downarrow$, i.e that Φ_a is total. That would mean $a \in TOT$, so there would be a number k such that $f(k) = a$. But then we would have $\Phi(k, a) = \Phi(k, f(k)) + 1 = \Phi(k, a) + 1$, a contradiction. Thus TOT is not re and, consequently, $TOT \not\leq_m K$. ■

THEOREM 6.2.3. *There are two unsolvable problems A and B which are incomparable, so that $A \not\leq_m B$ and $B \not\leq_m A$. Therefore, there are at least two incomparable non-recursive m-degrees.*

Proof. Take $A = K = \{x \in N \mid \Phi(x, x) \downarrow\}$ and $B = \overline{K} = \{x \in N \mid \Phi(x, x) \uparrow\}$. We first prove (a) $K \not\leq_m \overline{K}$, and then (b) $\overline{K} \not\leq_m K$.

(a) By contradiction. Assume that $K \leq_m \overline{K}$, so that there is a total Turing-computable function f such that $(\forall x)\ [x \in K \iff f(x) \in \overline{K}]$ (i). Now we know that K is re, so there must be a Turing-computable pnt function h such that $(\forall x)\ [x \in K \iff h(x)\downarrow]$ (ii). But then $\overline{K}$ would also have to be re, because there would exist a Turing-computable pnt function g, defined as $g(x) = h(f(x))$, such that $(\forall x)\ [x \in \overline{K} \iff g(x)\downarrow]$ (iii). This holds because (i) implies that for all x,

$$\begin{aligned}
[x \in K \iff f(x) \in \overline{K}] &\implies [x \notin K \iff f(x) \notin \overline{K}] \\
&\implies [x \in \overline{K} \iff f(x) \in K] \\
&\overset{(ii)}{\implies} [x \in \overline{K} \iff h(f(x))\downarrow] \\
&\implies [x \in \overline{K} \iff g(x)\downarrow].
\end{aligned}$$

We know, however, that $\overline{K}$ is not re, so we conclude that f does not exist and hence that $K \not\leq_m \overline{K}$.

(b) If $\overline{K} \leq_m K$, then by theorem 6.2.1.v, we would conclude that $\overline{K}$ is re. But that would be false, hence $\overline{K} \not\leq_m K$.[7] ■

[7] An alternative proof could use the preceding result (a) in tandem with theorem 6.2.1.iii.

It is worth remarking that TOT is not the only problem (from the ones we have seen so far) that is more unsolvable than K. It turns out that *FIN, CONST, INF*, and a few others too, are of a higher degree of unsolvability than the halting problem. We could prove these facts at this point by techniques similar to that used in the proof of the previous theorem. But if we did that we would be merely proving that *FIN, CONST, INF*, etc., are more m-unsolvable than K; and that would give us no information on how these problems stand *in relation to each other*, i.e as to whether FIN is more m-unsolvable than $CONST$, etc. To obtain such sharper knowledge we will have to wait until chapter 8. The methods developed there will enable us not only to prove in a much easier way that a certain problem A is more m-unsolvable than K, but also to (1) compute A's m-degree of unsolvability with precision, and thus (2) compare A with several unsolvable problems other than K.

6.3 One-one reducibility

One-one reducibility is a stronger form of m-reducibility, obtained by requiring the reduction function to be one-one. The basic ideas and definitions are the same:

DEFINITION 6.3.1. *A set A is termed* **1-reducible** *to a set B, written $A \leq_1 B$, iff there is a Turing-computable function f that is total, one-one, and such that $(\forall x)[\ x \in A \iff f(x) \in B\ ]$; we say that such an f "takes A into B". We write $A \equiv_1 B$ whenever $A \leq_1 B$ and $B \leq_1 A$, and we then say that A and B are equally 1-unsolvable. We say that B is more 1-unsolvable than A (or that A is less 1-unsolvable than B) iff $A \leq_1 B$ but $B \not\leq_1 A$. The equivalence classes of the $\equiv_1$ relation are called* **1-degrees**, *i.e the 1-degree of a set A is the class $deg_1(A) = \{B \subseteq N \mid A \equiv_1 B\}$. If* **a** *and* **b** *are 1-degrees, we write* $\mathbf{a} \preceq_1 \mathbf{b}$ *iff $A \leq_1 B$ for sets $A \in \mathbf{a}$ and $B \in \mathbf{b}$; we write* $\mathbf{a} \not\preceq_1 \mathbf{b}$ *otherwise. Finally, we say that* **a** *is* **lower** *than* **b** *(or that* **b** *is* **higher** *than* **a***) iff* $\mathbf{a} \preceq_1 \mathbf{b}$ *and* $\mathbf{a} \neq \mathbf{b}$*; we denote this by writing* $\mathbf{a} \prec_1 \mathbf{b}$.

The first question that we ought to consider with regard to this new reducibility is whether it coincides with m-reducibility, which amounts to asking whether two problems are 1-reducible to each other iff they are m-reducible. We will shortly answer this negatively, proving that the 1- and m-equivalence classes partition the power set of N in quite different ways. Moreover, in the next section we will prove that 1-reducibility enjoys a very special property that is not shared by its many-one counterpart. But for now let us study the structure of the 1-degrees and the order induced on

them by the $\prec_1$ relation, beginning, as usual, with the "solvable region": the recursive sets.

In the case of m-reducibility we originally suspected that all solvable problems would be trivially m-reducible to one another and that, as a result, there would be only one recursive degree, containing all and only the recursive sets. We saw that this was not entirely so due to N and $\emptyset$, so that all in all there were 3 recursive m-degrees. Owing to the ostensible similarity between m- and 1-reducibility, we might expect the same to be true for 1-reducibility, but it is not. The case of recursive 1-degrees is a little more complicated—it turns out there are infinitely many such degrees. For, consider any two finite sets of different cardinalities, say $A = \{0, 1\}$ and $B = \{5, 2, 9\}$. It is easily verified that (i) $A \leq_1 B$, but (ii) $B \not\leq_1 A$. To see (i), let

$$f(x) = \begin{cases} 5 & \text{if } x = 0 \\ 2 & \text{if } x = 1 \\ 6x & \text{if } x > 1. \end{cases}$$

By construction, f is a computable 1-1 function that takes A into B, i.e $(\forall x)[x \in A \iff f(x) \in B]$ (1). But no such f can take B into A, because B has an extra element, which, if taken into A, will make f many-one, and if not taken into A will fail to satisfy condition (1) (with A and B reversed). Thus we conclude that $B \not\leq_1 A$. Of course if A and B have the same cardinality, there is no problem; in that case there obviously exist computable bijections that will take A into B and vice versa, so that $A \equiv_1 B$. In summary:

THEOREM 6.3.1. *If A and B are finite then $A \equiv_1 B$ iff $\mid A \mid = \mid B \mid$. If $\mid A \mid < \mid B \mid$, then $A \leq_1 B$ but $B \not\leq_1 A$, so that $deg_1(A) \prec_1 deg_1(B)$.*

We can guess from this lemma that the lowest 1-degree will comprise the finite set with the smallest cardinality, namely the empty set $\emptyset$. The immediately higher degree will contain all the sets of cardinality one (i.e all the singletons $\{0\}, \{1\}, \{2\}, \ldots$). The next higher degree will comprise all and only the 2-element sets (such as $\{0, 956\}, \{5, 10^{27}\}$, etc.), and so forth.

In light of the above lemma, an arbitrary finite set A might or might not be 1-reducible to another finite set B, but we might incline to think that it ought to be 1-reducible to an arbitrary *infinite* set C. But that is not always true either. Specifically, it is true only if C is not co-finite, i.e only if it does not have a finite complement. For suppose $A = \{0, 1\}$ and C is infinite but co-finite, say $C = N - \{5\}$. Then to have $A \leq_1 C$ we must have a Turing-computable function f that is total, one-one and such that $(\forall x)[x \in A \iff f(x) \in C]$, which means that $x \in \overline{A} \implies f(x) \notin C \implies$

$f(x) \in \overline{C} \Longrightarrow f(x) = 5$, which means that f cannot be one-one (since $\overline{A}$ is infinite); thus $A \not\leq_1 C$. Similar considerations show that $C \not\leq_1 A$, so we conclude the following:

LEMMA 6.3.1. *If A is finite and B is co-finite, then A and B are 1-incomparable, i.e $A \not\leq_1 B$ and $B \not\leq_1 A$.*

It is true, however, that if B is both infinite and co-infinite then $A \leq_1 B$ (the reader can easily check this). We deduce

LEMMA 6.3.2. *If A is finite and B is infinite and co-infinite, then $A \leq_1 B$; the converse is false, i.e $B \not\leq_1 A$.*

We need just a few more similar results (all easily verifiable):

LEMMA 6.3.3. *If A and B are two co-finite sets, then $A \equiv_1 B$ iff $|\overline{A}| = |\overline{B}|$. Otherwise, if $|\overline{A}| < |\overline{B}|$, then $deg_1(A) \prec_1 deg_1(B)$, so that $A \leq_1 B$ but $B \not\leq_1 A$.*

LEMMA 6.3.4. *If A is co-finite and B is both infinite and co-infinite, then $deg_1(A) \prec_1 deg_1(B)$, so that $A \leq_1 B$ but $B \not\leq_1 A$.*

LEMMA 6.3.5. *If both A and B are infinite and co-infinite, then $A \equiv_1 B$, i.e $deg_1(A) = deg_1(B)$.*

We now put all the pieces together in fig. 6.4.

Thus we see that the m- and 1-reducibilities behave quite differently in the domain of solvable problems. What happens in the "unsolvable region" (which was appropriately left blank with questionmarks in fig. X)? Are the two relations co-extensive there? It would not be irrational to think so. After all, in the case of recursive sets the divergence of the two was basically due to cardinality differences, which is not an issue at all for unsolvable problems—since these are necessarily both infinite and co-infinite. Yet we will see that there are two non-recursive problems which are 1- but not m-equivalent, confirming that the two relations do indeed give rise to different degrees of *un*solvability. That will flow as an easy corollary from the answer to a different problem that we will tackle in chapter 8. For the time being we will discuss another important concept that turns out to be rather closely related to 1-equivalence: recursive isomorphism.

6.4 Recursive isomorphism and Myhill's first theorem

Let π stand for the Godel numbering scheme we devised in section 4.2 for the purpose of encoding Turing machines, so that for any machine M, $\pi(M) = \#(M) = [\#(I_1), \ldots, \#(I_n)]$, where the list $I_1, \ldots, I_n$ is the program of M (P_M), and $\#(I_j)$ is the godel number of instruction I_j. All of

Figure 6.4: Partition of the power-set of N by $\equiv_1$.

the important sets we have defined so far (*K*, *FINITE*, *TOTAL*, etc.) have been extensionally determined by the details of π, since these sets contain numbers that are construed as codes assigned to Turing machines by π. We have, for instance, $K = \{\pi(M) \mid$ machine M halts when started with its own godel number $\pi(M)$ as input $\}$. It follows that if we change the details of π, these sets would change as well.

So let π' be an encoding scheme other than π,[8] so that for most Turing machines M, $\pi(M) \neq \pi'(M)$ (of course we are assuming that π' is an *acceptable* arithmetization, i.e that is is one-one, onto, and mechanically performeable). And let K' be the set that encodes the halting problem in this new scheme, i.e $K' = \{\pi'(M) \mid$ machine M halts when started with its own code number $\pi'(M)$ as input $\}$. Now, as we have already explained, K and K' are two different sets, meaning that there are many numbers that are in K but not in K', and vice versa. From the viewpoint of computability theory, however, the two sets are indistinguishable because any recursion-theoretic property will hold for one of them iff it holds for the other. Take recursiveness, for example. It is obvious that K' is not recursive, because *if* it were recursive then so would K: to determine whether or not $x \in K$, for an arbitrary x, we would simply (1) follow the decoding algorithm of π to retrieve the machine M encoded by x (i.e M_x), (2) compute $\pi'(M)$ (the π' code of M), and then (3) check whether $\pi'(M) \in K'$. Or take the notion of recursive enumerability. It is clear that K' is re for exactly the same reasons for which K is re; neither could be re without the other being re as well.

The reason for this affinity between K and K' is that there exists a computable bijection between the two. Every number x in K has a unique counterpart number x' in K' (namely, the π' code of M_x), and conversely. More importantly, this bijection is mechanically computable: given an arbitrary x, we can find its counterpart x' by retrieving M_x and then computing $\pi'(M_x)$ (or, conversely, given an $x' \in K'$, we would retrieve M'_x and then compute $\pi(M'_x)$). To put it another way, every "object" in K has a unique "mirror object" in K' which can be found algorithmically, and vice versa. Therefore, the difference between the two is only "appearance-deep", so to speak, since we can move back and forth between the two representations (appearances) in a mechanical fashion. This ability allows us to strip away and ignore the difference in the *manner* in which the two sets encode their information, and focus on the information itself—which is essentially the

[8]There are many different ways to encode Turing machines. One choice for π' would be the Godel numbering scheme that is based on the fundamental theorem of arithmetic (this is actually the standard approach). For the details of such a method see Soare[X].

same: the halting problem. From a philosophical standpoint, one might say that K and K' have different *accidental* properties, but the same *essence*.

More specifically, the bijection between K and K' is established by the function $p(x) = \pi'(M_x)$, which returns the π' code of M_x (i.e the π' code of the machine whose π code is x). In other words, the lists

$$M_0, M_1, M_2, \ldots$$

$$M_{p(0)}, M_{p(1)}, M_{p(2)}, \ldots$$

are two identical enumerations of all Turing machines. We can think of p as a translation from the π representation to the π' representation. Note that p is (i) total (obvious), (ii) one-one (since $p(x) = p(y)$ for $x \neq y$ would imply that the encoding π' is not one-one), and (iii) onto N (since π' is onto); which means that p is a **permutation**[9]. And since p is mechanically computable, Church's thesis implies that (iv) p is Turing-computable. Now, as we have already explained, it is exactly the fact that p has properties (i)-(iv) that creates the said affinity between K and K'. So, generalizing, consider the family $\{p_i\}_{i=1}^{\infty}$ of all Turing-computable permutations (there is obviously a countable infinity of them). Then every set $K_i = \{p_i(x) \mid x \in K\}$, $i \in N$, has exactly the same relation to K that K' has: K_i and K encode the same information (the halting problem) under different guises. We say that K and K_i are **recursively isomorphic** and write $K \equiv K_i$. We extend the idea to arbitrary sets A and B as follows:

DEFINITION 6.4.1. *We say that two sets A and B are recursively isomorphic, written as $A \equiv B$, iff $B = \{ p_i(x) \mid x \in A\}$ for some Turing-computable permutation p_i.*

Informally, we say that p_i "translates" A into B. The fact that p_i is Turing-computable and a permutation ensures that the translation is legitimate (one-one, etc.) and mechanical.

Note that $\equiv$ is an equivalence relation: it is reflexive ($A \equiv A$ via the obviously computable identity permutation $i(x) = x$), it is symmetric (because the inverse of a computable permutation is the also computable permutation $p^{-1}(x) = min_z[z = p(x)]$, so that if $A \equiv B$ via p, we have $B \equiv A$ via p^{-1}), and finally it is transitive via function composition, which preserves computability.

[9] An nt function ρ that is total, one-one, and onto N is called a permutation because the list $\rho(0), \rho(1), \rho(2), \ldots$ can be viewed as a re-arrangement (permutation) of the natural numbers (the usual arrangement of course is $0, 1, 2, \ldots$, i.e the identity permutation $i(x) = x$).

The $\equiv$-equivalence class of a set A is the class of all sets that are recursively isomorphic to A, i.e the class comprising all $A_i = \{p_i(x) \mid x \in A\}$, $i \in N$; this class will be called the **recursive isomorphism type** of A, or ri-type of A for short. For example, the ri-type of K is the class of all the K_i sets that was mentioned above.

Ri-types are the "natural" objects of study in recursion theory, since all the sets that are in a single ri-type are "essentially the same"—which is why in advanced recursion theory sets are characterized "up to recursive isomorphism". This means that we are only interested in those properties of sets which are preserved under computable permutations. Such properties are called **recursively invariant**. In other words, a set-theoretic property P is recursively invariant if, for any given ri-type T, either all sets in T have P or none do. Recursiveness and recursive enumerability are two examples of recursively invariant properties. An example of a property that is not recursively invariant is that of containing some particular number(s), say, the number three. For consider the sets $A = \{3, 5\}$ and $B = \{5, 17\}$, and the computable permutation

$$p(x) = \begin{cases} x & \text{if } x \neq 3 \text{ and } x \neq 17 \\ 17 & \text{if } x = 3 \\ 3 & \text{if } x = 17. \end{cases}$$

Then, clearly, $A \equiv B$ since $B = \{p(x) \mid x \in A\}$, yet A contains 3 whereas B does not. Another set-theoretic property that is not recursively invariant is that of containing the even numbers.

This approach is actually found in many other theoretical fields. In Topology, for example, we study spaces "up to homeomorphism", meaning that we are concerned mostly with those properties of topological spaces that are preserved under homeomorphisms, i.e with "homeomorphically invariant" properties. Likewise, in Euclidean geometry we study figures "up to displacement", while in Logic and in abstract algebra we study models of first-order theories and algebraic structures, respectively, "up to isomorphism". This is the "space transformation" approach, introduced in mathematics by Klein: a branch of mathematics studies the subspaces of a certain space (set), which comes equipped with a set of "similarity transformations", which are total bijections from the space onto itself; then any two subspaces A and B are considered, in the particular theory at hand, to be indistinguishable (or rather, *similar*) if $f(A) = B$, where f is a similarity transformation (see Tourlakis[1984], on recursive isomorphism). In the theory of computation, the space is N and the similarity transformations are

the computable permutations. Rogers[1967] is the author who introduced the Klein approach to computability theory.

Figure 6.5: $A \equiv_1 B$ does not impy $A \equiv B$.

It should be pointed out that, although it might seem so, $A \leq_1 B$ does *not* imply $A \equiv B$, because the function that takes A into B might not be *onto* B. A simple example of such a situation is illustrated in fig. 6.5. However, $A \leq_1 B$ *and* $B \leq_1 A$ does imply $A \equiv B$, as the following theorem[10] confirms.

MYHILL'S FIRST THEOREM *Recursive isomorphism and 1-equivalence coincide, i.e* $A \equiv B \Longleftrightarrow A \equiv_1 B$.

Proof. Half of the left-to-right part is immediate. The other half is also trivial given that the inverse of a computable permutation is another com-

[10]The content and proof of this theorem (which was given by Myhill in 1955) are almost direct adaptations of the classic Cantor-Schroder theorem in Set theory, which establishes that if two sets can be 1-1 mapped into each other, then they must be equinumerous.

putable permutation.

In the other direction, suppose that $A \equiv_1 B$, so that there are two Turing-computable functions f and g which are total, one-one, and such that for all x, $x \in A \Longleftrightarrow f(x) \in B$ **(I)**, and $x \in B \Longleftrightarrow g(x) \in A$ **(II)**. Now to prove that $A \equiv B$, we must show that there is a function h such that:
(i) h is total,
(ii) h is onto N,
(iii) h is one-one, and
(iv) for all x, $x \in A \Leftrightarrow h(x) \in B$.

Our proof will be *constructive*, i.e we will actually construct h rather than merely deduce its existence. Of course, as a function, h is an infinite set and we can never quite "construct" an infinite object. What we can, however, effectively construct, are finite but *arbitrarily long segments* of h, i.e arbitrarily long finite lists of pairs of numbers. The longer the lists the closer we get to having constructed h in full. We will finally define h to be the limit of these lists as their length approaches infinity. This might sound a little vague at this point, but it will become perfectly clear as we begin to fill in the details.

In particular, our construction will proceed in stages $s = 1, 2, 3, ...$ In the initializing stage $s = 1$ we construct a list L_1 with only one pair of numbers, i.e $L_1 = \{< x_1, y_1 >\}$, in such a way that $x_1 \in A \Leftrightarrow y_1 \in B$. In stage $s = 2$ we add one more pair to L_1 to obtain $L_2 = \{< x_1, y_1 >, < x_2, y_2 >\}$; x_2 and y_2 are chosen so that $x_2 \neq x_1, y_2 \neq y_1$, and $x_2 \in A \Leftrightarrow y_2 \in B$. In stage $s = 3$ we add to L_2 one more pair $< x_3, y_3 >$ that satisfies similar constraints, and so forth. We finally set

$$h = \bigcup_{s=0}^{\infty} L_s$$

so that for any list $L_s = \{< x_1, y_1 >, ..., < x_s, y_s >\}$, we have $h(x_i) = y_i$ $(1 \leq i \leq s)$. Despite the fact that it is given by means of a limiting process, this definition is constructive. For, given an arbitrary $n \in N$, we can easily compute $h(n)$ by going through the various successive stages of the construction algorithm until we generate a pair $< n, m >$ (the algorithm will make sure that such a pair is eventually generated); the answer will then simply be $h(n) = m$. So h is clearly mechanically computable so that, by Church's thesis, it is also Turing-computable. In the course of the discussion it will become evident that requirements **(i) - (iv)** are also satisfied.

We now give a high-level description of how the algorithm works, without getting too much into details. For any list $L_s = \{< x_1, y_1 >, ..., < x_s, y_s >\}$ let us agree to write D_s for the set $\{x_1, \ldots, x_s\}$ (the "domain" of L_s) and

R_s for $\{y_1, ..., y_s\}$ (the "range" of L_s). Now after the initial stage $s = 1$ in which we set $L_1 = \{< 0, f(0) >\}$, the algorithm repeatedly alternates between two courses of action, depending on whether the current stage s is even or odd. Recall that in all stages $s > 1$ we have a list $L_{s-1} = \{< x_1, y_1 >, \ldots, < x_{s-1}, y_{s-1} >\}$ satisfying the construction requirements, and by adding a new pair $< x_s, y_s >$ to this list we produce a new list L_s that also satisfies these requirements. In odd stages s, the new pair $< x_s, y_s >$ is obtained as follows: we let x_s be the least number not yet in the domain of the list, i.e not in D_{s-1} (towards satisfying **(i)**), and we try to find an appropriate value for y_s, i.e a y_s such that
(a) $x_s \in A \Leftrightarrow y_s \in B$ (towards satisfying **(iv)**), and
(b) $y_s \neq y_i, 1 \leq i \leq s-1$ (towards satisfying **(iii)**).
So, in odd stages we "know" x_s and we try to find an appropriate match y_s. In even stages, we let y_s be the least number not yet in the range of the list (towards satisfying **(ii)**), and we try to find an appropriate value for x_s, i.e an x_s such that
(a) $x_s \in A \Leftrightarrow y_s \in B$ (towards satisfying **(iv)**), and
(b) $x_s \neq x_i, 1 \leq i \leq s-1$ (since h must be a function).
So, in even stages we "know" y_s and we try to find x_s. As was already hinted, we split the process into odd and even stages for the purpose of satisfying **(i)** and **(ii)**. To understand this, note that if in odd stages s we take the *least* number i not yet in the domain of the list (i.e not in D_{s-1}) and *we put it in the domain* by explicitly setting $x_s = i$, then *every* number $i \in N$ will eventually appear in D_s, for sufficiently advanced stages s; and that, in turn, means that

$$\bigcup_{s=1}^{\infty} D_s = N$$

i.e that h is total. Likewise, if in even stages s we make sure that the least number i not yet in the range of the list (i.e not in R_{s-1}) is forced into the range by explicitly setting $y_s = y$, then for sufficiently large s *any* number $i \in N$ will appear in R_s, so that

$$\bigcup_{s=1}^{\infty} R_s = N$$

and h will be onto N.

Finally, here is the algorithm in detail:

Initializing stage s = 1

We let $x_1 = 0$ and $y_1 = f(0)$, i.e $L_1 = \{< 0, f(0) >\}$, so that $h(0) = f(0)$**(1)**. Clearly, $0 \in A \overset{\textbf{(I)}}{\Longleftrightarrow} f(0) \in B \overset{\textbf{(1)}}{\Longleftrightarrow} h(0) \in B$.

Stage s > 1

We assume we have a list $L_{s-1} = \{< x_1, y_1 >, ..., < x_{s-1}, y_{s-1} >\}$, such that $x_i \in A \iff y_i \in B$ **(2)**, $x_i \neq x_j$ **(3)**, and $y_i \neq y_j$, $1 \leq i < j \leq s-1$. This is our inductive hypothesis. We will now add one more pair $< x_s, y_s >$ to obtain L_s. There are two cases:

- s **is odd:** Let $x_s = min_i[i \notin D_{s-1}]$. We must find a value y_s such that
  (a) $y_s \neq y_i$, $1 \leq i \leq s-1$ (since h must be 1–1), and
  (b) $x_s \in A \iff y_s \in B$ (to satisfy **(iv)**).
  We try $y_s = f(x_s)$. If $f(x_s)$ satisfies **(a)**, i.e if $f(x_s) \neq y_i$ for $i = 1, 2, ..., s-1$, then we are done because **(b)** is certainly satisfied: $x_s \in A \overset{\textbf{(I)}}{\iff} f(x_s) \in B$. If, however, $f(x_s) = y_j$ **(5)** for some j in $\{1, 2, \ldots, \text{s-1}\}$, then we must find another value for y_s. We try $y_s = f(x_j)$. Then if $f(x_j)$ satisfies **(a)** we are done, since **(b)** is also satisfied: $x_s \in A \overset{\textbf{(I)}}{\iff} f(x_s) \in B \overset{\textbf{(5)}}{\iff} y_j \in B \overset{\textbf{(2)}}{\iff} x_j \in A \overset{\textbf{(I)}}{\iff} f(x_j) \in B$. Now if $f(x_j)$ is in R_{s-1}, i.e if $f(x_j) = y_m$ **(6)** for some m in $\{1, 2, ..., s-1\}$, we proceed in the same manner, i.e we try $y_s = f(x_m)$, check to see if $f(x_m)$ is in R_{s-1} (if it is not we are done since $x_s \in A \iff f(x_s) \in B \overset{\textbf{(5)}}{\iff} y_j \in B \overset{\textbf{(2)}}{\iff} x_j \in A \overset{\textbf{(I)}}{\iff} f(x_j) \in B \overset{\textbf{(6)}}{\iff} y_m \in B \overset{\textbf{(2)}}{\iff} x_m \in B \overset{\textbf{(I)}}{\iff} f(x_m) \in B$), and so forth. The procedure is bound to terminate because at least one of $f(x_s), f(x_1), f(x_2), ..., f(x_{s-1})$ is not in $\{y_1, y_2, ..., y_{s-1}\}$. This follows from (A) the fact that $x_s \neq x_1 \neq x_2 \neq ... \neq x_{s-1}$, (B) the fact that f is 1-1 (for different arguments it has different values), and (C) the pigeonhole principle.

- s **is even:** We follow a perfectly symmetrical procedure. In particular, we let $y_s = min_i[i \notin R_{s-1}]$ and we set out to find a value x_s such that
  (a) $x_s \neq x_i$, $1 \leq i \leq s-1$ (since h must be a function), and
  (b) $x_s \in A \iff y_s \in B$ (to satisfy **(iv)**.
  We try $x_s = g(y_s)$**(7)**. If $g(y_s)$ satisfies **(a)** then we are done, since **(b)** is satisfied as follows: $y_s \in B \overset{\textbf{(II)}}{\iff} g(y_s) \in B \overset{\textbf{(7)}}{\iff} x_s \in B$. If, however, we have $g(y_s) = x_j$ **(8)** for some j in $\{1, 2, ..., s-1\}$, we must find another value for x_s. We try $x_s = g(y_j)$**(9)**. If $g(y_j) \notin R_{s-1}$, we are done since $y_s \in B \overset{\textbf{(II)}}{\iff} g(y_s) \in A \overset{\textbf{(8)}}{\iff} x_j \in A \overset{\textbf{(2)}}{\iff} y_j \in B \overset{\textbf{(II)}}{\iff} g(y_j) \in A \overset{\textbf{(9)}}{\iff} x_s \in A$, otherwise we proceed in the same fashion. The procedure is guaranteed termination since not all of $g(y_s), g(y_1), g(y_2), ..., g(y_{s-1})$ can be in $\{x_1, x_2, ..., x_{s-1}\}$ (from the

pigeonhole principle, since (A) $y_s \neq y_1 \neq y_2 \neq ... \neq y_{s-1}$, and (B) g is 1-1).

Inspection of the algorithm should make it evident that defining h as $\bigcup_{s=1}^{\infty} L_s$ meets all requirements **(i)** − **(iv)**. As an interesting sidelight, the reader might want to verify the following corollary: for each $i \in N$, the pairs $< i, h(i) >$ and $< h^{-1}(i), i >$ will appear in the list before the stage $2i + 2$. We conclude the proof with a pseudocode version of the above algorithm:

(* Initializing stage s = 1 *)
$s \longleftarrow 1$;
$x_1 \longleftarrow 0$;
$y_1 \longleftarrow f(0)$;
$s \longleftarrow s + 1$;
Stages s > 1
WHILE (True) DO BEGIN
  IF Odd(s) THEN BEGIN
    $x_s \longleftarrow min_i[i \notin D_{s-1}]$;
    $y_s \longleftarrow f(x_s)$;
    $Done \longleftarrow False$;
    WHILE (NOT Done) DO BEGIN
      $y_s\_Found\_In\_Range \longleftarrow False$;
      $i \longleftarrow 0$;
      REPEAT
        IF $(y_s = y_i)$ **THEN**
          $y_s\_Found\_In\_Range \longleftarrow True$
        ELSE
          $i \longleftarrow i + 1$;
      UNTIL $[\ (i > s - 1)$ **OR** $(y_s\_Found\_In\_Range)]$;
      IF $y_s\_Found\_In\_Range$ **THEN**
        $y_s \longleftarrow f(x_i)$
      ELSE
        $Done \longleftarrow True$;
    END; **(* While NOT Done *)**
  END **(* Odd stage *)**
  ELSE BEGIN **(* Even stage*)**
    $y_s \longleftarrow min_i[i \notin R_{s-1}]$;
    $x_s \longleftarrow g(y_s)$;
    $Done \longleftarrow False$;
    WHILE (NOT Done) DO BEGIN
      $x_s\_Found\_In\_Domain \longleftarrow False$;

```
            i ⟵ 0;
            REPEAT
                IF (x_s = x_i) THEN
                    x_s_Found_In_Domain ⟵ True;
                ELSE
                    i ⟵ i + 1;
            UNTIL [ (i > s − 1) OR (x_s_Found_In_Domain) ];
            IF x_s_Found_In_Domain THEN
                x_s ⟵ g(y_i)
            ELSE
                Done ⟵ True;
        END;      (* WHILE (NOT Done) DO BEGIN *)
    END;      (* Even stage *)
    s ⟵ s + 1;
END;      (* While (True) DO BEGIN *)
```

■

We can now prove the following:

THEOREM 6.4.2. *The sets* $K = \{x \in N \mid \Phi(x, x) \downarrow\}$ *and* $KH = \{x \in N \mid \Phi(l(x), r(x)) \downarrow\}$ *are recursively isomorphic.*

Proof. We will prove that $K \equiv_1 KH$; the result will then follow from the previous theorem. First we show that $K \leq_1 KH$. We let $f(x) =< x, x >$ and observe that f is total, one-one, and Turing-computable. Furthermore, $(\forall x)\ [x \in K \iff f(x) \in KH]$, thus $K \leq_1 KH$.

In the other direction, consider the following algorithm: given a number x, construct a Turing machine M_a such that for all inputs z, $\Phi_a(z) = \Phi_{r(x)}(l(x))$. Thus the behavior of M_a is independent of its input z—M_a invariably simulates $M_{r(x)}$ on input $l(x)$. Therefore, Φ_a is either the nowhere-defined function (if $\Phi_{r(x)}(l(x)) \uparrow$), or the constant function $h(x) = \Phi_{r(x)}(l(x))$ (if $\Phi_{r(x)}(l(x)) \downarrow$). Thus $(\forall z)\ [\Phi_a(z) \downarrow \iff \Phi(l(x), r(x)) \downarrow]$, and for $z = a$,

$$\Phi_a(a) \downarrow \iff a \in K \iff \Phi(l(x), r(x)) \downarrow \iff x \in KH. \quad (1)$$

By Church's thesis, there is a Turing machine M_n that implements this algorithm, so that $\Phi_n(x)$ produces the said godel number a. Hence (1) implies that for all x, $x \in KH \iff \Phi_n(x) \in K$. Moreover, because godel numbers are unique, Φ_n is one-one. And because the above algorithm produces an output a for all inputs x, Φ_n is total. Thus Φ_n takes KH into K and establishes that $KH \leq_1 K$. ■

6.5 Complete sets, creative sets, productive sets

Let **Q** be a collection of sets of numbers. We say that a set $C \in \mathbf{Q}$ is **m-complete for Q** iff $A \leq_m C$ for all sets $A \in \mathbf{Q}$. Intuitively, C is the "hardest" or "most unsolvable" problem (with respect to m-reducibility) in the entire collection **Q**. This does not necessarily imply that C is unique, for **Q** might contain many m-complete sets; i.e there might be another set $D \neq C$ in **Q** such that $A \leq_m D$ for all $A \in \mathbf{Q}$ (and thus $C \leq_m D$ as well). What it does imply is that there is nothing in **Q** that is *more* m-unsolvable than C; i.e there is no set $D \in \mathbf{Q}$ such that $C \leq_m D$ *and* $D \not\leq_m C$. Likewise, we say that a set $C \in \mathbf{Q}$ is **1-complete for Q** iff $A \leq_1 C$ for all sets A in **Q**. Henceforth we will take **Q** to be the class of re sets. Accordingly, we will be using the terms 'm-complete' and '1-complete' without qualification, tacitly understanding the completeness to be with respect to the re sets.

What is the significance of complete sets? Well, knowing that first-order theories are representeable as re sets, it follows that the decision problem of *any* first-order theory is reducible to the computation of any 1- or m-complete set. To put it another way:
If we could compute an 1-complete set (or an m-complete set), then all first-order theories would be decideable.

We begin by showing that K is 1-complete (and hence m-complete as well):

THEOREM 6.5.1. *K is 1-complete, i.e for all re sets A, $A \leq_1 K$.*

Proof. Since A is re, there is a Turing machine M_a that halts for all and only the elements of A, i.e $A = W_a = \{x \in N \mid \Phi_a(x)\downarrow\}$. Now consider the following algorithm: given a number x, construct a Turing machine M_b such that for all inputs z, $\Phi_b(z) = \Phi_a(x)$. Thus the behavior of M_b is independent of its input z; no matter what input we inscribe on its tape, M_b will invariably run M_a on x. It follows that $(\forall x)\Phi_b(z)\downarrow \iff \Phi_a(x)\downarrow$. Let b, the godel number of the constructed machine, be the output of the algorithm. Note that the algorithm
(i) halts and produces an output b for all inputs x, and
(ii) for two different inputs x and x', it produces two different outputs b and b' (since different Turing machines have different godel numbers).

By Church's thesis, there exists a Turing machine M_k that carries out this algorithm. By remarks (i) and (ii), Φ_k is total and one-one. Furthermore, we claim that $(\forall x)\ [x \in A \iff \Phi_k(x) \in K]$. For, if $x \in A$, then $\Phi_a(x)\downarrow$, so (1) implies that $\Phi_{\Phi_k(x)}(\Phi_k(x))\downarrow$ (remember that $\Phi_k(x)$ is the godel number b), i.e $\Phi_k(x) \in K$; and if $x \notin A$, then $\Phi_a(x)\uparrow$, so again by (1), $\Phi_{\Phi_k(x)}(\Phi_k(x))\uparrow$, i.e $\Phi_k(x) \notin K$. Thus Φ_k proves that $A \leq_1 K$. ■

The following is a simple but very useful fact:

THEOREM 6.5.2. *Suppose A and B are two re sets, and suppose that A is 1-complete. Then $A \leq_1 B$ implies that B is 1-complete too.*

Proof. For any re set E, we have $E \leq_1 A$ (since A is 1-complete). And since $A \leq_1 B$, the transitivity of $\leq_1$ implies that $E \leq_1 B$. ■

Recall that in proving K and KH to be recursively isomorphic (theorem 6.4.2), we showed that $K \leq_1 KH$. Therefore, the above theorem implies that

THEOREM 6.5.3. *$KH = \{x \in N \mid \Phi_{r(x)}(l(x)) \downarrow\}$ is 1-complete.*

Interpretation: *If we could compute the halting problem, we could solve the decision problem of any first-order theory.*

Theorem 6.5.3 is a formal verification of the corresponding intuitive argument that we gave in the previous part, which we now recapitulate: for any first-order theory T, let A_T be the re set of the encoded theorems of T (for any acceptable encoding). Since A is re, there is a Turing machine M_a such that $A_T = W_a = \{x \in N \mid \Phi_a(x) \downarrow\}$. Now to find out whether an arbitrary sentence is a theorem of T, i.e whether an arbitrary number x is in A_T, simply check whether $< x, a > \in KH$. If $< x, a > \in KH$, then $\Phi_a(x) \downarrow$, and x is a theorem of T; if $< x, a > \notin KH$, then $\Phi_a(x) \uparrow$ and x is not a theorem of T. That's the power of the halting problem.

A set P is called **productive** if there is a Turing-computable pnt function f such that $(\forall x)\ [W_x \subseteq P \Longrightarrow f(x) \downarrow$ and $f(x) \in P - W_x]$; f is said to be a "productive function" for P. A productive set cannot be re. Indeed, a productive set P can be *mechanically shown to be non-re*: for suppose someone claimed that P is re, i.e that $W_x = P$ (1) for some $x \in N$. Then there is an algorithm (namely, the Turing program that computes the productive function f of P) which will *produce* a counter-example falsifying (1)—namely, the number $f(x)$, which will be in P but not in W_x. In fact, this strongly demonstrable non-recursive-enumerability is exactly what the notion of productiveness[11] was intended to capture. We summarize:

THEOREM 6.5.4. *Productive sets are not re.*

The quintessential productive set is $\overline{K}$. Here the productive function is the identity $i(x) = x$. For, suppose $W_x \subseteq \overline{K}$ (1). Then $i(x) = x$ cannot be in W_x, because $x \in W_x \Longrightarrow \Phi(x, x) \downarrow \Longrightarrow x \in K \Longrightarrow x \notin \overline{K}$ (2), while (1) implies that $x \in W_x \Longrightarrow x \in \overline{K}$ (3). Thus (2) and (3) prove that $x \in W_x$ is a contradictory assumption, and we conclude that $x \notin W_x$. But if $x \notin W_x$, then $\Phi(x, x) \uparrow$, so that $x \in \overline{K}$. Therefore, $W_x \subseteq \overline{K}$ implies that $i(x) = x \in \overline{K} - W_x$.

[11] Introduced by Dekker.

A set A is called **creative** iff it is re and $\overline{A}$ is productive. Just as the quintessential productive set is $\overline{K}$, the paradigmatic creative set is K. The following is an immediate consequence of the previous theorem:

THEOREM 6.5.6. *A creative set is not recursive.*

Proof. For A to be recursive, both it and $\overline{A}$ must be re. But $\overline{A}$ is productive, so, by the previous theorem, it cannot be re. ■

As we have alsready seen in the previous part, the set of theorems of ENT (elementary number theory) is creative. Indeed, most "interesting" theories have creative sets of theorems, as the complements of such sets ($\{$ unprovable sentences$\}$ $\cup$ $\{$ refutable sentences $\}$) are productive. This, in tandem with the previous theorem, shows that mathematics cannot be done mechanically, i.e that there is no algorithm for discovering theorems. In other words, mathematical thinking requires *creativity*. That's exactly what Post intended to convey through the use of the term 'creative'.[12] We conclude this section with two interesting results that will come handy later on.

THEOREM 6.5.7. *If A is productive, then it has a total productive function.*

Proof. We are assuming that A is productive, i.e that there is a Turing-computable pnt function τ such that

$$(\forall x)\ [W_x \subseteq A \Longrightarrow \tau(x)\downarrow\ \wedge \tau(x) \in A - W_x]\ \ (i),$$

and we are seeking to replace τ with a *total* Turing-computable function t. Consider the following algorithm: given an arbitrary number z, we construct a machine M_k which, given an input number x, does the following two things: (1) First it checks to see whether or not $\tau(z)\downarrow$ (this first step is independent of the input x). Since τ might be non-total, this check might never finish (if $\tau(z)\uparrow$). (2) If it does finish (i.e if $\tau(z)\downarrow$), then the second and last step of M_k is to start simulating M_z on input x, i.e it begins the computation $\Phi_z(x)$ (which also might or might not converge). This algorithm works for any number z, so, by Church's thesis, there must exist a *total* Turing-computable function $g(z) = k$, where k is the godel number of the machine constructed by the algorithm.

By the definition of g, it follows that for any number z we have $\tau(z)\downarrow \Longrightarrow (\forall x)\ [\Phi(x, g(z)) = \Phi(x, z)]$, while $\tau(z)\uparrow \Longrightarrow (\forall x)\ \Phi(x, g(z))\uparrow$. In other words,

$$W_{g(z)} = \begin{cases} W_z & \text{if } \tau(z)\downarrow \\ \emptyset & \text{if } \tau(z)\uparrow. \end{cases} \quad (ii)$$

[12] Post introduced that term in his 1944 paper "Re sets of positive integers and their decision problems".

It now follows that $(\forall z)$ [either $\tau(z)\downarrow$ or $\tau(g(z))\downarrow$] (iii). For if $\tau(z)\downarrow$, then (iii) is clearly true, while if $\tau(z)\uparrow$, then (ii) implies that $W_{g(z)}=\emptyset$ and hence $W_{g(z)}\subseteq A$, so that (i) yields $\tau(g(z))\downarrow$. We finally define $t(z)$ as either $\tau(z)$ or $\tau(g(z))$, whichever halts in fewer steps ((iii) guarantees that one of them definitely halts). That is, to obtain $t(z)$ for some $z\in N$, we start dovetailing the computations of $\tau(z)$ and $\tau(g(z))$ until one of them halts with a certain output value. By (iii), t is total, and, by Church's thesis, it is Turing-computable. To see that it is also a productive function for A, suppose that we have $W_z\subseteq A$ (iv), for an arbitrary z. Then (i) implies that $\tau(z)\downarrow$ (v) and $\tau(z)\in A-W_z$ (vi). But (ii) and (v) together imply that $W_{g(z)}=W_z$ (vii), so now (vii) and (iv) imply that $W_{g(z)}\subseteq A$, which means that $\tau(g(z))\downarrow$ and $\tau(g(z))\in A-W_z$ (viii). From (vi) and (viii) we conclude that both $\tau(z)$ and $\tau(g(z))$ belong to $A-W_z$. But $t(z)$ will be either $\tau(z)$ or $\tau(g(z))$, hence $t(z)\in A-W_z$. Since the argument is valid for arbitrary z, it follows that t is a total Turing-computable productive function for A. ■

THEOREM 6.5.8. *If A is productive, it has an one-one productive function.*

Proof. We have a total productive function t and we want to make it into a total *one-one* function k. Consider the following algorithm: given a $z\in N$, construct a Turing machine M_a which, given an input number x, does the following two things. First it computes $t(z)$ and checks to see if $t(z)=x$. If this is true, it halts, otherwise it takes its second and last step, which is to start simulating M_z on x, i.e it begins the computation $\Phi_z(x)$. By Church's thesis, there exists a total Turing-computable function $f(z)=a$, where a is the godel number of the constructed machine. Hence, for any number z, we have $(\forall x)$ $[\Phi(x,f(x))\downarrow\iff(x=t(z))\ \vee\ \Phi_z(x)\downarrow]$, or, equivalently, $W_{f(z)}=W_z\cup\{t(z)\}$ (1). Note that $(\forall z)$ $[W_z\subseteq A\Longrightarrow W_{f(z)}\subseteq A]$ (2), because if $W_z\subseteq A$ (3), then $t(z)\in A$ (4) (since t is a productive function for A), hence by (1), (3), and (4), we see that $W_{f(z)}\subseteq A$.
Now for a given $z\in N$ define the following sequence of numbers:

$$b_0=z$$
$$b_{i+1}=f(b_i)\quad i=1,2,3,\ldots$$

By virtue of (2), a simple induction can show that $W_z\subseteq A\Longrightarrow W_{b_i}\subseteq A$ (5), for all $i\in N$. Also notice that, for all i, $W_{b_{i+1}}=W_{f(b_i)}=W_{b_i}\cup\{t(b_i)\}$ (6).

Now assume that $W_z\subseteq A$. Then (5) implies that for any $i\in N$, $W_{b_{i+1}}\subseteq A$, which means that $t(b_{i+1})\in A-W_{b_{i+1}}$, hence $t(b_{i+1})\notin W_{b_{i+1}}$ (7). But, by (6), $t(b_i)\in W_{b_{i+1}}$ (8), so that (7) and (8) together tell us that $t(b_i)\neq t(b_{i+1})$. This is true for all i, so we can conclude that for all z,

If $W_z \subseteq A$, *then* the numbers $t(b_0), t(b_1), t(b_2), \ldots$ are distinct. (9)

We can now define k constructively. We let $k(0) = t(0)$, and for $z > 0$ we let $D_z = \{k(y) \mid y < z\}$. We must now define k so that $k(z) \notin D_z$ (since k must be one-one). To do this, we use the following algorithm: we start computing the numbers $t(b_0), t(b_1), t(b_2), \ldots$, until *either* (i) a repetition occurs, *or* (ii) a number $t(b_\lambda)$ is generated that is not in D_z. One of the two conditions must eventually become true, because either all of the numbers $t(b_0), t(b_1), \ldots$ are distinct, in which case (ii) will be satisfied since D_z is finite, or else there is one or more repetition, in which case (i) will be true. Now if our computation stops due to (i), then by (9) and Modus Tollens we know that $W_z \not\subseteq A$, so nothing is demanded of the value $k(z)$ in the way of productivity; thus we simply let $k(z)$ be the least number not in D_z, to ensure the one-oneness and computability of k. On the other hand, if (ii) becomes true, then we define $k(z) = t(b_\lambda)$. Then if $W_z \subseteq A$, we have (from (5)) $W_{b_\lambda} \subseteq A$ and thus $t(b_\lambda) \in A - W_{b_\lambda}$ (10). But (6) tells us that $W_z \subseteq W_{b_\lambda}$, hence (10) implies that $t(b_\lambda) = k(z)$ is in $A - W_z$. Therefore, k is a total one-one productive function for A (and, of course, k is Turing-computable by construction). ■

6.6 Myhill's second theorem

In this section we will prove that 1-completeness, m-completeness, creativity, and K-isomorphism are all co-extensive properties! This is a deep result (the proof is quite sophisticated) with many interesting repercussions in the subject of first-order theories. Its full significance will be discussed in the next section, in connection with Post's problem.

We begin with a few technical preliminaries. Let us call a predicate $P(x, y_1, \ldots, y_k)$ re if there is a Turing-computable predicate $Q(x, y_1, \ldots, y_k, z)$ such that for all $x, y_1, \ldots, y_k$,

$$P(x, y_1, \ldots, y_k) \iff (\exists z)\, Q(x, y_1, \ldots, y_k, z).$$

THEOREM 6.6.1. *If $P(x, y_1, \ldots, y_k)$ is an re predicate, then there is a total Turing-computable function $g : N^k \longrightarrow N$ that is one-one and such that $P(x, y_1, \ldots, y_k) \iff \Phi_{g(y_1,\ldots,y_k)}(x) \downarrow$. Equivalently,*
$P(x, y_1, \ldots, y_k) \iff x \in W_{g(y_1,\ldots,y_k)}$.

Proof. Since P is re, there is a Turing-computable predicate $Q : N^{k+2} \longrightarrow \{0, 1\}$ such that $P(x, y_1, \ldots, y_k) \iff (\exists z)\, Q(x, y_1, \ldots, y_k, z)$. And since Q is

Turing-computable, there is a machine M_q such that

$$\Phi_q(x, y_1, \ldots, y_k, z) = \begin{cases} 1 & \text{if } Q(x, y_1, \ldots, y_k, z) \\ 0 & \text{if } \neg Q(x, y_1, \ldots, y_k, z). \end{cases}$$

Now consider the following algorithm: given k numbers $y_1, \ldots, y_k$, construct a machine M_a such that for all inputs x, $\Phi_a(x) = min_z[\Phi_q(x, y_1, \ldots, y_k, z) = 1] = min_z[Q(x, y_1, \ldots, y_k, z)]$. Clearly,

$$\Phi_a(x) \downarrow \iff (\exists z)\, Q(x, y_1, \ldots, y_k, z) \iff P(x, y_1, \ldots, y_k). \quad (1)$$

Let the output of the algorithm be a, i.e the godel number of the constructed machine.

By Church's thesis, there must be a Turing machine M_b that carries out this algorithm. Thus, for any $y_1, \ldots, y_k$, $\Phi_b^k(y_1, \ldots, y_k)$ will be the godel number a of formula (1). Since godel numbers are unique, Φ_b is one-one, i.e $\Phi_b^k(y_1, \ldots, y_k) = \Phi_b^k(y_1', \ldots, y_k') \implies y_i = y_i'$ for $i = 1, \ldots, k$. And since the algorithm we gave halts and outputs an appropriate a for all k-tuples $y_1, \ldots, y_k$, Φ_b^k is total. Furthermore, (1) implies that for all $x, y_1, \ldots, y_k$, $\Phi_{\Phi_b^k(y_1,\ldots,y_k)}(x) \downarrow \iff P(x, y_1, \ldots, y_k)$; therefore, Φ_b^k is the Turing-computable total and 1-1 function we were looking for. ■

Another result that we will need is the so-called **parameterized recursion theorem** (due to Kleene):

THEOREM 6.6.2. *If $f(x, y)$ is a total Turing-computable function, then there is a total one-one Turing-computable function $g(y)$ such that for all x and y, $\Phi(x, g(y)) = \Phi(x, f(g(y), y))$.*

Proof. Consider the following algorithm: given two numbers x and y, construct a machine M_a such that for all z, $\Phi_a(z) = \Phi(z, \Phi_x^2(x, y))$ (1). Informally, M_a first tries to compute $\Phi_x^2(x, y)$. If a result r is eventually obtained, it is treated as a godel number and the computation of $\Phi(z, r)$ commences (which, in turn, might or might not halt). Otherwise, (i.e if $\Phi_x^2(x, y) \uparrow$), M_a obviously diverges on z. Let the output of the algorithm be the godel number a.

By Church's thesis, there is a Turing machine M_k that carries out this algorithm, so that $\Phi_k^2(x, y)$ will be the godel number a of equation (1). Thus $\Phi_{\Phi_k^2(x,y)}(z) = \Phi(z, \Phi_x^2(x, y))$ (2). Now, because godel numbers are unique, Φ_k^2 is one-one; and, because the algorithm halts for all x and y, Φ_k^2 is total. Next, construct a Turing machine M_s such that $\Phi_s^2(x, y) = f(\Phi_k^2(x, y), y)$ (3), and let $g(y) = \Phi_k^2(s, y)$ (4). Note that both Φ_s^2 and g are total and

Turing-computable (since f and Φ_k^2, respectively, are such). Moreover, for all y and z, we have

$$\begin{aligned}\Phi_{g(y)}(z) &= \Phi(z, \Phi_k^2(s,y)) \\ &= \Phi(z, \Phi_s^2(s,y)) \\ &= \Phi(z, f(\Phi_k^2(s,y), y)) \\ &= \Phi(z, f(g(y), y)). \quad \blacksquare\end{aligned}$$

We can now prove the following:

THEOREM 6.6.3. *If A is productive, then $\overline{K} \leq_1 A$.*

Proof. Let τ be a total and one-one productive function for A, so that $(\forall n)\,[W_n \subseteq A \Longrightarrow \tau(n) \in A - W_n]$ (1). We will show that there is a total Turing-computable function ξ that is one-one and such that $(\forall y)\,[y \in \overline{K} \Longleftrightarrow \xi(y) \in A]$ (2), i.e that $\overline{K} \leq_m A$. Consider the predicate

$$P(x,y,z) \Longleftrightarrow \Phi(z,z)\downarrow \ \wedge (\tau(y) = x) \quad (3).$$

We can also express this as $P(x,y,z) \Longleftrightarrow (\exists t)\,[STEP(z,z,t) \wedge (\tau(y) = x)]$, and since the expression within the square brackets is Turing-computable[13], P is re. Therefore, by theorem 6.6.1, we have $P(x,y,z) \Longleftrightarrow \Phi(x, f(y,z))\downarrow$ (4), for some total and 1-1 Turing-computable f. In other words, for all x, y, and z, we have $\Phi(z, f(x,y))\downarrow \Longleftrightarrow [(y \in K) \wedge (\tau(x) = z)]$ (5), which implies that

$$\begin{aligned} &y \in K \Longrightarrow [\Phi(z, f(x,y))\downarrow \Longleftrightarrow \tau(x) = z] \quad \text{, while} \\ &y \notin K \Longrightarrow \Phi(z, f(x,y))\uparrow. \end{aligned}$$

Or, equivalently, that

$$W_{f(x,y)} = \begin{cases} \tau(x) & \text{if } y \in K \\ \emptyset & \text{if } y \notin K. \end{cases} \quad (6)$$

Applying the parameterized recursion theorem to the function f, we infer that there is a total, one-one, and Turing-computable function g such that for all x and y, $\Phi(x, f(g(y), y)) = \Phi(x, g(y))$, so that $W_{g(y)} = W_{f(g(y),y)}$, and by (6),

$$W_{g(y)} = \begin{cases} \tau(g(y)) & \text{if } y \in K \\ \emptyset & \text{if } y \in \overline{K}. \end{cases} \quad (7)$$

We now define the desired function as $\xi(y) = \tau(g(y))$. Since both τ and g are total and Turing-computable, the same must hold for ξ. Now if y is an

[13] Note that the equality $\tau(y) = x$ is Turing-computable since τ is total.

arbitrary number, we have
$y \in \overline{K} \stackrel{(7)}{\Longrightarrow} W_{g(y)} = \emptyset \Longrightarrow W_{g(y)} \subseteq A \Longrightarrow \tau(g(y)) \in A \Longrightarrow \xi(y) \in A$ (8), while
$y \notin \overline{K} \stackrel{(7)}{\Longrightarrow} W_{g(y)} = \{\tau(g(y))\} \Longrightarrow W_{g(y)} = \{\xi(y)\}$ (9).
Now assume that $W_{g(y)} \subseteq A$. Then, from (1), we get $\tau(g(y)) = \xi(y) \in A - W_{g(y)}$, i.e $\xi(y) \notin W_{g(y)}$. But that contradicts (9), so we conclude that $W_{g(y)} \not\subseteq A$ (10).
But now (9) $\stackrel{(10)}{\Longrightarrow} \xi(y) \notin A$ (11). Finally, (9) and (11) together prove (2). ■

THEOREM 6.6.4. *If A is m-complete, then A is creative.*
Proof. We will show that there is a total Turing-computable function τ such that $(\forall n)[W_n \subseteq \overline{A} \Longrightarrow \tau(n) \in \overline{A} - W_n]$ (1), i.e that $\overline{A}$ is productive—and hence that A is creative. Consider the re set $KH = \{< x, y > | \Phi(x, y) \downarrow\}$. Since A is m-complete, we must have $KH \leq_m A$, i.e there must be a total Turing-computable function f such that $(\forall x)[x \in KH \Longleftrightarrow f(x) \in A]$. In other words, for all x and y we have

$$\Phi(x, y) \downarrow \Longleftrightarrow < x, y > \in KH \Longleftrightarrow f(< x, y >) \in A \quad (2).$$

Note that the contrapositive of (2) yields $f(< x, y >) \in \overline{A} \Longleftrightarrow \Phi(x, y) \uparrow$ (3).

Now consider the binary predicate $P(x, n) \Longleftrightarrow \Phi(f(< x, x >), n) \downarrow$. We can also express this as $P(x, n) \Longleftrightarrow (\exists t)\ [STEP(f(< x, x >), n, t)]$ (I), and since $STEP, f$, and $<, >$ are all Turing-computable, the expression within the square brackets of (I) is Turing-computable and P is re. Consequently, theorem 6.6.1 implies that there is a Turing-computable function g such that $P(x, n) \Longleftrightarrow \Phi(x, g(n)) \downarrow$, i.e for all x and n we have $\Phi(f(< x, x >), n) \downarrow \Longleftrightarrow \Phi(x, g(n)) \downarrow$ (4). We now define the desired function as

$$\tau(n) = f(< g(n), g(n) >).$$

Since both f and g are total and Turing-computable, so is τ. To see that τ also satisfies (1), assume that for some number n_0 we have $W_{n_0} \subseteq \overline{A}$ (5). Then we must have $\Phi(g(n_0), g(n_0)) \uparrow$ (6). To see this, assume $\neg(6)$, i.e assume that $\Phi(g(n_0), g(n_0)) \downarrow$ (7). Then

$$\begin{aligned}
(7) &\stackrel{(4)}{\Longrightarrow} \Phi(f(< g(n_0), g(n_0) >), n_0) \downarrow \\
&\Longrightarrow f(< g(n_0), g(n_0) >) \in W_{n_0} \\
&\stackrel{(5)}{\Longrightarrow} f(< g(n_0), g(n_0) >) \in \overline{A} \\
&\Longrightarrow \Phi(g(n_0), g(n_0)) \uparrow \\
&\Longrightarrow \neg(7), \text{a contradiction.}
\end{aligned}$$

Thus we conclude (6). But now (4) and (6) together imply that

$\Phi(f(< g(n_0), g(n_0) >), g(n_0)) \uparrow$, i.e $f(< g(n_0), g(n_0) >) = \tau(n_0) \notin W_{g(n_0)}$ (8).

And, by (6) and (2) we conclude that $f(< g(n_0), g(n_0) >) = \tau(n_0) \in \overline{A}$ (9). Finally, (8) and (9) together imply that $\tau(n_0) \in \overline{A} - W_{n_0}$. Since the proof was given for arbitrary n_0, it follows that τ is a productive function for A. ■

We finally reach our goal:
MYHILL'S SECOND THEOREM. *For any re set A, the following are equivalent:*
(i) A is 1-complete
(ii) A is m-complete
(iii) A is creative
(iv) A and K are recursively isomorphic.
Proof. We show that $(i) \Longrightarrow (ii) \Longrightarrow (iii) \Longrightarrow (i)$ and that $(i) \Longleftrightarrow (iv)$. Since 1-complete sets are trivially m-complete, the $(i) \Longrightarrow (ii)$ part is immediate. The $(ii) \Longrightarrow (iii)$ part is given by the previous theorem. To show that $(iii) \Longrightarrow (i)$: let A be creative, so that $\overline{A}$ is productive. From theorem 6.6.3, we have $\overline{K} \leq_1 \overline{A}$, hence, from theorem 6.2.1.iii, $K \leq_1 A$. But since we know K to be 1-complete (theorem 6.5.1), theorem 6.5.2 implies that A is 1-complete. Thus we have shown that conditions *(i)–(iii)* are equivalent. Now suppose that A is 1-complete. Then $K \leq_1 A$, and since K is also 1-complete, $A \leq_1 K$. Hence $A \equiv_1 K$ and Myhill's first theorem entails that $A \equiv K$. Conversely, suppose that $A \equiv K$. Then $K \leq_1 A$ and theorem 6.5.2 implies that A is 1-complete. Therefore, $(i) \Longleftrightarrow (iv)$, and from our work in the previous paragraph, $(i) \Longleftrightarrow (ii) \Longleftrightarrow (iii) \Longleftrightarrow (iv)$. ■

It follows from part (iv) that
COROLLARY 6.6.1. *All creative sets are recursively isomorphic.*

6.7 Post's problem (many-one version)

All of the re sets we have seen so far have been one of two kinds: either they were recursive or they were 1-complete. So the question naturally arises: are these the *only* two kinds of re sets? In other words, are all non-recursive re sets necessarily 1-complete?—and hence, by Myhill's second theorem, m-complete and creative and isomorphic to K as well? Or is there some

"middle ground", i.e some re sets that are non-recursive, yet not "quite as non-recursive" as the halting problem? This is Post's problem *with respect to 1-completeness*. Fig. 6.6 is a graphical formulation of it.

Of course the significance of Post's problem is better understood when one keeps in mind that re sets encode first-order theories; and, consequently, that the non-existence of certain types of re sets entails the non-existence of the corresponding types of first-order theories. For example, decidable first-order theories are encoded by recursive re sets, while undecidable first-order theories give rise to non-recursive re sets. Therefore, the non-existence of certain types of non-recursive re sets automatically implies the non-existence of certain types of undecidable first-order theories. Bearing this correlation in mind, we give a precise statement of Post's problem along with its interpretation from the perspective of mathematical theories:

Post's problem: *Is there a non-recursive re set A that is not 1-complete, i.e such that $K \not\leq_1 A$? In other words, is there a non-recursive re set that is less 1-unsolvable than the halting problem*?

Interpretation: *Is there an undecidable first-order theory which, even if it were decidable, the halting problem would still be unsolvable?*

Here are some other equivalent formulations:

(i) Is there a non-recursive re set A that is not recursively isomorphic to the halting problem?

(ii) Is there an 1-degree $\mathbf{a}$ *such that* $\mathbf{INFREC} \prec_1 \mathbf{a} \prec_1 deg_1(K)$, *where* **INFREC** *is the 1-degree of all the infinite and co-infinite recursive sets?*

(iii) Are all non-recursive re sets creative?

In virtue of the aforementioned interpretation, we can conclude the following:

A negative solution *to Post's problem would entail that all undecidable theories are of the same 1-degree of undecidability as the halting problem, and thus that they all have recursively isomorphic sets of theorems. It would then follow that if any undecidable first-order theory were decidable, the halting problem would be solvable.*

A positive solution *to Post's problem would entail that some undecidable theories are not "quite as undecidable" as number theory or ZF. Even if such theories were decidable, the halting problem would still be unsolvable.*

Furthermore, and more interestingly, a positive solution would imply that there are some undecidable theories in which we cannot "talk about" the halting problem, or, more precisely, in which we cannot **represent** the set $K = \{x \in N \mid \Phi_x(x) \downarrow\}$—or any set isomorphic to it. In particular, let us say that a set A is **representeable** in a theory T iff there is a wff $\phi(v)$ in the language of T, with exactly one free variable, such that for all numbers

x, $x \in A \Longleftrightarrow \vdash \phi(\overline{x})$, where $\overline{x}$ is some ground term that denotes x.
THEOREM 6.7.1. *If the set K, or any other creative set, is representeable in a first-order theory T, then the theorems of T form an 1-complete set.*
Proof. If K is representable in T, then there is a wff $\phi(v)$ in the language of T such that for all x, $x \in K \Longleftrightarrow \vdash \phi(\overline{x})$ (1). Now let A_T be the set of the code numbers of T's theorems (under any suitable 1-1 algorithmic encoding of the wffs of T's language), and for any $x \in N$, let $f(x)$ be the code number of the wff $\phi(\overline{x})$. Then (1) implies that $x \in K \Longleftrightarrow f(x) \in A_T$ (2), and since f is total, 1-1, and Turing-computable, (2) implies that $K \leq_1 A_T$, and thus A_T is 1-complete. The same argument will work for any creative set in place of K. ■

Now a positive solution to Post's problem would mean that there are undecidable theories whose theorems do *not* form 1-complete sets; then the contrapositive of the above theorem would imply that sets isomorphic to K are *not* representeable in such theories. We summarize:
A positive solution *to Post's problem would entail that there are some undecidable theories in which we cannot represent the halting problem, or any set that is recursively isomorphic to the halting problem; i.e in such theories we would not be able to represent any creative sets.*

When his problem was originally posed, Post (correctly) conjectured that it had a positive solution, and he explicitly set out to *construct* a set that would settle the issue affirmatively. What he was looking for, of course, was an re set that was neither recursive nor creative. His strategy was straight-forward: first he discovered a property P that all creative sets necessarily have, and then he "constructed" an re set A that did *not* have P—so that it couldn't be creative. Since A also turned out to be non-recursive, his question was conclusively settled. Now what is this "property P" that creative sets must have? It is very simple: the complement of a creative set *must* contain an infinite re subset. Since the complement of a creative set is, by definition, a productive set, the above amounts to saying that productive sets have infinite re subsets. The following is a proof of that.
THEOREM 6.7.2. *A productive set has an infinite re subset.*
Proof. Consider the following algorithm: given a number $x = [a_1, \ldots, a_k]$, use the decoding algorithm of section 4.2 to retrieve the $k > 0$ numbers $a_1, \ldots, a_k$ encoded by x. Then construct a Turing machine M_a that halts only for the numbers $a_1, \ldots, a_k$, i.e such that $W_a = \{a_1, \ldots, a_k\}$. In rough outline, we would design M_a to do the following: given an input z, it would compare z successively to $a_1, \ldots, a_k$, i.e to $(x)_1, \ldots, (x)_{LEN(x)}$. If an equality

is discovered, M_a would immediately halt. Otherwise, if z is found to be distinct from all a_i, $i = 1, \ldots, k$, then M_a would get into an infinite loop. Thus

$$(\forall z)\ \Phi(z,a)\downarrow \iff [z = (x)_1 \vee z = (x)_2 \vee \cdots \vee z = (x)_{LEN(x)}] \quad (1),$$

and $W_a = \{a_1, \ldots, a_k\} = \{(x)_i \mid i = 1, \ldots, LEN(x)\}$ (2). Let the output of the algorithm be a, the godel number of the constructed machine. By Church's thesis, this algorithm is implemented by some Turing machine M_s, so that $\Phi_s(x)$ is the godel number a of equation (1). Thus (1) implies that

$$(\forall z)\ [\Phi(z, \Phi_s(x))\downarrow \iff [z = (x)_1 \vee \cdots \vee z = (x)_{LEN(x)}] \quad (3),$$

so that $W_{\Phi_s(x)} = \{(x)_i \mid i = 1, \ldots, LEN(x)\}$ (4). Now let P be a productive set, so that $(\forall x)\ [W_x \subseteq P \Longrightarrow f(x) \in P - W_x]$ (5), for some total Turing-computable f. We will define a total Turing-computable function h in such a way that the set $A = \{h(x) \mid x \in N\}$ will be an infinite subset of P. Since A will be the range of a total Turing-computable function (namely, h), it will be re (by definition), and the proof will be complete. Specifically, let p be any element of P and define h as

$$h(0) = p$$
$$h(n+1) = f(\Phi_s([h(0), \ldots, h(n)])).$$

LEMMA. *For all $i \in N$, (i) $h(i) \in P$, and (ii) $h(i) \notin \{h(0), \ldots, h(i-1)\}$.*
Proof. We use strong induction for (i) and simple induction for (ii), simultaneously. For $i = 0$ (ii) holds vacuously, while (i) holds because $p \in P$. Now assume that for some $i \in N$,
(INDi) $h(j) \in P$ for all $j = 0, 1, \ldots, i$, and
(INDii) $h(i) \notin \{h(0), \ldots, h(i-1)\}$.
From (4) we get $W_{\Phi_s([h(0),\ldots,h(i)])} = \{h(0), \ldots, h(i)\}$ (6), so, from (INDi), $W_{\Phi_s([h(0),\ldots,h(i)])} \subseteq P$. Thus (5) yields $f(\Phi_s([h(0), \ldots, h(i)])) = h(i+1) \in P$, and (i) is proved by strong induction. But (5) also implies that

$$f(\Phi_s([h(0), \ldots, h(i)])) = h(i+1) \notin W_{\Phi_s([h(0),\ldots,h(i)])},$$

thus (6) $\Longrightarrow h(i+1) \notin \{h(0), \ldots, h(i)\}$, and (ii) also follows by induction. ■

Part (ii) establishes that $A = \{h(i) \mid i \in N\}$ is infinite, while part (i) shows that A is a subset of P. Finally, the result follows because h is total and Turing-computable, and thus its range A is re. ■

Now let us call a set A **simple** iff it is re and such that $\overline{A}$ is infinite but contains no infinite re subset. By the above theorem, a simple set cannot be

creative. And it cannot be recursive either, as $\overline{A}$ would then be an infinite re subset of $\overline{A}$. Therefore, being re, A would be a positive answer to Post's problem. Of course the crucial question now is: *are there any simple sets*? Post answered this question affirmatively with a direct construction, which we present in the proof of the following theorem.

THEOREM 6.7.3. *Simple sets exist.*

Proof. Recall that the re sets are all and only the sets $W_i = \{x \in N \mid \Phi_i(x) \downarrow\}$, $i \in N$. We define a simple set S constructively, by means of the following procedure: start dovetailing the enumerations of $W_0, W_1, W_2, \ldots$ (see the endnote for one possible way of doing that). Put into S the first discovered member of W_i (if any) that is greater than $2i$. More formally,

$$S = \{x \in N \mid (\exists i)\ (\exists t)\ [STEP(x,i,t) \ \wedge \ x > 2i \ \wedge \ (\forall y < x)[y > 2i \Longrightarrow \neg STEP(y,i,t)]]\}.$$

LEMMA 1. *S intersects every infinite re set, i.e $S \cap W_i \neq \emptyset$ for all infinite W_i.*

Proof. If W_i is infinite, then the set $A = \{x > 2i \mid \Phi_i(x) \downarrow\}$ is non-empty. Thus the above dovetailing procedure will eventually discover some element x of A and put it in S, so that $S \cap A \neq \emptyset$, and since $A \subseteq W_i$, $S \cap W_i \neq \emptyset$. ■

LEMMA 2. *For each $i \geq 0$, at most i of the numbers $\{0, 1, \ldots, 2i\}$ are elements of S.*

Proof. If one of the numbers $0, 1, \ldots, 2i$ *is* an element of S, it must have come from one of the i sets $W_0, W_1, \ldots, W_{i-1}$—because for $n \geq i$, W_n can only contribute numbers $> 2i$. (For example, for $i = 5$, if one of the numbers $\{0, 1, \ldots, 10\}$ is in S, it must have come from one of W_0, W_1, W_2, W_3, W_4, since W_n for $n \geq 5$ can only yield numbers $> 2 \cdot 5 = 10$). But each of the i sets $W_0, \ldots, W_{i-1}$ can contribute *at most* one element to S, so even if all of them contribute numbers in the $\{0, 1, \ldots, 2i\}$ range, no more than i such contributions can be made. ■

We can now easily prove the following:

(1) *$\overline{S}$ has no infinite re subset.* For suppose A was such a subset. Then, by the first lemma, there is a number x such that (a) $x \in S$, and (b) $x \in A$. But since $A \subseteq \overline{S}$, (b) $\Longrightarrow$ $x \in \overline{S}$, contradicting (a).

(2) *$\overline{S}$ is infinite.* For, lemma 2 implies that for each i, *at least* $i + 1$ of the numbers $\{0, 1, \ldots, 2i\}$ are not elements of S, i.e they are elements of $\overline{S}$.

And, since S is re (by construction), it follows that it is simple.

Note The following is one way to dovetail the enumerations of all the re

sets:

Stage 1: Compute 1 step of $\Phi_0(0)$.

Stage 2: Compute 2 steps of $\Phi_0(0), \Phi_0(1), \Phi_1(0)$.

Stage 3: Compute 3 steps of $\Phi_0(0), \Phi_0(1), \Phi_0(2), \Phi_1(0), \Phi_1(1), \Phi_2(0)$.

$\vdots$

Stage n: Compute n steps of $\Phi_0(0), \ldots, \Phi_0(n-1), \Phi_1(0), \ldots, \Phi_1(n-2), \ldots, \Phi_{n-2}(0), \Phi_{n-2}(1), \Phi_{n-1}(0)$.

Note that the number c_n of n-step computations performed at stage n is given by the non-homogeneous recurrence relation

$$c_1 = 1$$
$$c_{n+1} = c_n + n.$$

■

Chapter 7

Computing with oracles

Although the $\leq_m$ and $\leq_1$ relations seem to be natural and fairly accurate formalizations of the intuitive notion of reducibility, they have one serious shortcoming. Intuitively, every set A should be reducible to its complement $\overline{A}$; for, *if* we could compute $\overline{A}$, the computation of A would be trivial: for any given x, we would let $c_A(x) = 1$ if $c_{\bar{A}}(x) = 0$ and $c_A(x) = 0$ if $c_{\bar{A}}(x) = 1$. Yet this is not always the case with m-reducibility. Consider, for example, any non-recursive re set, say $K = \{x \in N \mid x \in W_x\}$. If we had $K \leq_m \overline{K}$ it would follow that $\overline{K} \leq_m K$ and thus that $\overline{K}$ is re (theorem 6.2.1.v), which we know is not true. Hence $K \not\leq_m \overline{K}$, and since $\leq_1$ is a restriction of $\leq_m$, we also have $K \not\leq_1 \overline{K}$—contrary to what intuition would have us believe.

This pitfall is avoided by another type of reducibility, introduced by Turing and named after him, which is widely considered to be the most natural of all known reducibilities and which forms the theoretical basis of uncomputability. Turing-reducibility is based on the notion of an **oracle**, which is a fictional device that can provide answers to questions of the form "$x \in S$?" for arbitrary x and S, even for non-recursive S. Informally, we say that a set A is Turing-reducible to a set B, written $A \leq_T B$, if, given an arbitrary $x \in N$, we can determine whether or not $x \in A$ in a finite amount of time *provided* we have access to a "B-oracle" which can answer any question of the form "$y \in B$?", for any number y. It is thus straightforward to see that $A \leq_T \overline{A}$ for any set A. It is also readily seen that the $\leq_1$ and $\leq_m$ reducibilities are restrictions of $\leq_T$, i.e that $A \leq_1 B \Longrightarrow A \leq_m B \Longrightarrow A \leq_T B$. For, assuming that $A \leq_1 B$ or $A \leq_m B$ via some Turing-computable function f, determining whether $x \in A$ boils down to asking the B-oracle one question: is $f(x)$ in B? This will be made more precise after Turing-reducibility has been given a formal definition.

7.1 Oracle Turing machines

Our definition of Turing machines will have to be slightly modified to incorporate these new ideas. The new machines will be called **oracle Turing machines** and will be symbolized by the same letters as the regular Turing machines but with hats on them: $\hat{M}, \hat{M}_1, \hat{M}'$, etc. Now an oracle Turing machine $\hat{M}$ (fig. 7.1) differs from an ordinary Turing machine in the following respects:
(1) It has two tapes instead of one. One of them is the customary work tape, while the other is the so-called "oracle tape". The cells of the latter contain either 0s or 1s, signifying the values of the characteristic function of some set A, called **the oracle of** $\hat{M}$. In particular, the symbol written on the x^{th} square of the oracle tape (counting from left to right) signifies the value of $c_A(x-1)$, so that the first (leftmost) cell gives us the value of $c_A(0)$, the second gives us the value $c_A(1)$, etc. Simply put, we can find out whether or not a number x is in A by consulting the $(x+1)^{st}$ cell of the oracle tape. We assume for simplicity that the two tapes are left-aligned.
(2) As a result of the added tape, the processor now has two scanning devices: a read/write head for the work tape, and a read-only head for the oracle tape. These two heads always scan cells that are vertically aligned.
(3) The program of $\hat{M}$, denoted $P_{\hat{M}}$, is a list of *six-tuples* $q_i \;\; s \;\; s' \;\; q_j \;\; s'' \;\; m$, where q_i, q_j are states of $\hat{M}$, $s, s'' \in \{1, \#\}, s' \in \{0, 1\}$, and $m \in \{L, R\}$. Such an instruction is interpreted as follows: if $\hat{M}$ is in state q_i and the current squares of the work and oracle tapes contain s and s', respectively, then $\hat{M}$ changes states to q_j, overwrites the s on the work tape with s'', and moves one square to the left or to the right, depending on the value of m.

The execution semantics of oracle Turing machines are similar to the ordinary semantics: we begin a computation by inscribing $k > 0$ input numbers $x_1, \ldots, x_k$ on the work tape of $\hat{M}$ (in the usual format: x_1 consequtive 1s on the left end, then a blank, then x_2 consequtive 1s, and so on). Then execution proceeds in the customary fashion, with the topmost instruction of the form $q_1 \;\; 1 \;\; s' \;\; q_j \;\; s'' \;\; m$. If the "halting state" q_0 is ever reached, or if $P_{\hat{M}}$ has no instructions of the appropriate form, then the computation stops and we define $\psi_{\hat{M}}^{A,k}(x_1, \ldots, x_k)$, **the output of** $\hat{M}$ **on input(s)** $x_1, \ldots, x_k$ **and with oracle** A, to be the total number of 1s left on the work tape.[1] Otherwise we say that $\hat{M}$ **diverges on input(s)** $x_1, \ldots, x_k$ **with oracle** A, and we write $\psi_{\hat{M}}^{A,k}(x_1, \ldots, x_k) \uparrow$.[2] Sometimes the output of $\hat{M}$ on $x_1, \ldots, x_k$

[1] As before, the superscript k is ommitted when $k = 1$.

[2] Of course the fact that $\hat{M}$ does halt on $x_1, \ldots, x_k$ with oracle A is expressed as

Figure 7.1: An oracle Turing machine.

with oracle A is also called the **A-output of** $\hat{M}$ **on** $x_1, \ldots, x_k$. Finally, here is the important definition: a pnt function $f : N^k \longrightarrow N$ is called **A-computable**, or **computable in** A, iff there is an oracle Turing machine $\hat{M}$ with oracle A such that for all numbers $x_1, \ldots, x_k$,

(1) $f(x_1, \ldots, x_k) \uparrow \Longleftrightarrow \psi_{\hat{M}}^{A,k}(x_1, \ldots, x_k) \uparrow$, and

(2) $f(x_1, \ldots, x_k) \downarrow \Longrightarrow \psi_{\hat{M}}^{A,k}(x_1, \ldots, x_k) \downarrow$, and $f(x_1, \ldots, x_k) = \psi_{\hat{M}}^{A,k}(x_1, \ldots, x_k)$.

In other words, f is A-computable iff there is an oracle Turing machine $\hat{M}$ *with* c_A *written on its oracle tape*, which (i) gets into an infinite loop for all and only those numbers $x_1, \ldots, x_k$ for which f is undefined, and (ii) converges and outputs the value $f(x_1, \ldots, x_k)$ for all numbers $x_1, \ldots, x_k$ for which f is defined.

It is important to realize that one and the same machine $\hat{M}$ will yield

$\psi_{\hat{M}}^{A,k}(x_1, \ldots, x_k) \downarrow$.

different outputs for the same input number(s) depending on what its oracle is, i.e depending on what is written on its oracle tape. In the case of a regular Turing machine M, $\psi_M(x)$, for example, is a specific, fixed function from N to N. But for an oracle Turing machine $\hat{M}$, $\psi^A_{\hat{M}}(x)$ can be *any one* of an uncountable collection of functions from N to N, depending on which one of the uncountably many infinite sets of numbers A stands for. Thus, for instance, if $A \neq B$, $\psi^A_{\hat{M}}(5)$ could very well be different from $\psi^B_{\hat{M}}(5)$, despite the fact that the same program ($P_{\hat{M}}$) is executed with the same input (5).

Our next step is to arithmetize oracle Turing machines. We use the same ideas as before. We define a finite bijection $\hat{C} : \{1, \#\} \times \{0, 1\} \times \{1, \#\} \times \{L, R\} \longrightarrow \{0, 1, \ldots, 15\}$, which assigns a unique code number (in the $0 \ldots 15$ range) to any combination of s, s', s'', and m in an instruction $I : q_i \;\; s \;\; s' \;\; q_j \;\; s'' \;\; m$. Then we define the **godel number** $\#(I)$ of such an instruction to be the number $< i, 16j + \hat{C}(s, s', s'', m) >$. Finally, we define the **godel number of** $\hat{M}$ to be the number $\#(\hat{M}) = [\#(I_1), \ldots, \#(I_n)]$, where $I_1, \ldots, I_n$ are the instructions of $P_{\hat{M}}$. This scheme is readily seen to be 1-1, onto, and mechanically computable. Note that the number zero again becomes the godel number of the "empty" oracle Turing machine. As before, we write $\hat{M}_x$ for the oracle Turing machine with godel number x. Thus the list $\hat{M}_0, \hat{M}_1, \ldots$ is an enumeration of all and only the oracle Turing machines. We write $\hat{P}_x$ for the program of $\hat{M}_x$; $\hat{P}_x$ and $\hat{M}_x$ will be used interchangeably, since, for all practical purposes, an oracle Turing machine can be identified with its program.[3] Finally, we define

$$\Phi^{A,k}(x_1, \ldots, x_k, y) = \Phi^{A,k}_y(x_1, \ldots, x_k) = \psi^{A,k}_{\hat{M}_y}(x_1, \ldots, x_k), \quad \text{and}$$

$$STEP^{A,k}(x_1, \ldots, x_k, y, t) \Longleftrightarrow [\hat{M}_y \;\; \text{with } A \text{ on its oracle tape halts on } x_1, \ldots, x_k \;\; \text{in } t \text{ or fewer steps}].$$

When we only have one input argument we drop the superscript $k = 1$ and write $\Phi^A(x, y)$ or $\Phi^A_y(x)$, and $STEP^A(x, y, t)$.

Church's thesis implies the following:

Relativized Church's thesis: *If a pnt function f is intuitively computable in a mechanical fashion with the aid of an oracle that answers questions of the form "$x \in A$?", for some set A, then f is A-computable.*

Now if we have an A-oracle, the function $\Phi^{A,k}(x_1, \ldots, x_k, y)$ is certainly mechanically computable in the intuitive sense: given $x_1, \ldots, x_k, y$, first decode y and retrieve $\hat{P}_y$. Then query the oracle and find out the values of an initial segment of c_A, say $c_A(0), \ldots, c_A(100)$, and write these values on

[3] Of course for computational purposes we would also need to know the oracle of $\hat{P}_x$.

the first 101 squares of the oracle tape. Then start executing $\hat{P}_y$ in accordance with the aforementioned rules. If the scanning devices ever attempt to move to the right of the 101^{st} square, query the oracle, determine the values $c_A(101), \ldots, c_A(200)$, write them on the appropriate oracle squares, and continue the computation in the same fashion. A similar argument can show that $STEP^{A,k}(x_1, \ldots, x_k, y, t)$ can be mechanically computed with the aid of an A-oracle. Therefore, the relativized version of Church's thesis implies the following:

THEOREM 7.1.1. *The functions* $\Phi^{A,k}(x_1, \ldots, x_k, y)$ *and* $STEP^{A,k}(x_1, \ldots, x_k, y, t)$ *are A-computable.*

Finally, note that a *total* function f can also serve as an oracle in the form of the set $\{< x, f(x) >| \ x \in N\}$. This is what we will mean when we write $\Phi_y^{f,k}(x_1, \ldots, x_k)$ or $STEP^{f,k}(x_1, \ldots, x_k, y, t)$. We emphasize that f must be total, for otherwise no finite amount of querying the oracle $\{< x, f(x) >| \ x \in N\}$ can ever determine whether or not $f(x)$ has a value *if* in fact it does not.

7.2 Turing reducibility

We can now define Turing reducibility with precision. The basic ideas are the same as in the $\leq_m$ and $\leq_1$ cases. We say that a set A is **Turing-reducible** to a set B, written $A \leq_T B$, iff c_A is computable in B. The complement of the $\leq_T$ relation is denoted by $\not\leq_T$. We write $A \equiv_T B$ to signify that $A \leq_T B$ and $B \leq_T A$. It is readily seen that $\equiv_T$ is an equivalence relation; its equivalence classes are called **Turing degrees**. We continue to denote degrees by boldface lower-case letters such as **a**, **b**, etc. We define a partial order $\preceq_T$ on Turing degrees as follows: $\mathbf{a} \preceq_T \mathbf{b} \iff A \leq_T B$ for some $A \in \mathbf{a}$ and $B \in \mathbf{b}$. We say that a degree **a** is **lower than** a degree **b** (or that "**b** is higher than **a**"), denoted $\mathbf{a} \prec_T \mathbf{b}$, iff $\mathbf{a} \preceq_T \mathbf{b}$ and $\mathbf{a} \neq \mathbf{b}$. In such a case we say that the problems in **a** are "less T-unsolvable" than those in **b**. The symbol $\succ_T$ will denote the inverse of $\prec_T$, i.e $\mathbf{b} \succ_T \mathbf{a}$ iff $\mathbf{a} \prec_T \mathbf{b}$.

THEOREM 7.2.1. $A \leq_1 B$ *implies* $A \leq_m B$ *implies* $A \leq_T B$.

Proof. That $A \leq_1 B$ implies $A \leq_m B$ is trivial. Now suppose there is a total Turing-computable function f such that $x \in A \iff f(x) \in B$. Then to compute $c_A(x)$ with the aid of a B-oracle, simply compute $f(x)$ (this must halt since f is total), and then ask the oracle whether $f(x) \in B$. Thus the relativized form of Church's thesis implies that $A \leq_T B$. ■

THEOREM 7.2.2. *If A and B are recursive, then* $A \equiv_T B$.

Proof. Since A is recursive, we have $A \leq_T B$ because we can compute $c_A(x)$

with or without a B-oracle. The same goes for $B \leq_T A$, hence $A \equiv_T B$. ■

The proofs of the next two theorems are very easy and we ommit them.

THEOREM 7.2.3. *For all sets A, $A \equiv_T \overline{A}$.*

THEOREM 7.2.4. *If A is recursive and B is not, then $A \leq_T B$ but $B \not\leq_T A$.*

Theorems 7.2.2 and 7.2.4 imply that there is a Turing degree which contains all and only the recursive sets, and which is the lowest of all Turing degrees. It is customary to denote that degree with the symbol $\mathbf{0}$.

Finally, we call an re set C **Turing-complete** iff $A \leq_T C$ for all re sets A. The transitivity of $\leq_T$ implies the following.

THEOREM 7.2.5. *If C is Turing-complete and A is an re set such that $C \leq_T A$, then A is Turing-complete.*

Since we have proved that the set K is 1-complete, theorem 7.2.1 implies that K is Turing-complete. Furthermore, if $A \equiv K$ then $K \leq_1 A$ (by Myhill's first theorem), so $K \leq_T A$ (theorem 7.2.1) and thus A is Turing-complete. We summarize these facts as follows:

THEOREM 7.2.6. *The halting problem K and all sets that are recursively isomorphic to it (i.e all creative—1-complete—m-complete sets) are Turing-complete.*

It is conventional to denote the Turing degree of K with the symbol $\mathbf{0}'$. Since a set A in $\mathbf{0}'$ is $\equiv_T$-equivalent to K (by definition), it follows that $K \leq_T A$ and that A is Turing-complete. Therefore, $\mathbf{0}'$ contains all Turing-complete re sets, but not *only* those. For example, since $K \equiv_T \overline{K}$, we have $\overline{K} \in \mathbf{0}'$—and $\overline{K}$ is not even re. We conclude:

THEOREM 7.2.7. *The Turing degree $\mathbf{0}'$ contains all Turing-complete sets, and hence, by the previous theorem, all creative sets as well; but it also contains many other different sets. In particular, $\mathbf{0}'$ contains sets that are not re, such as $\overline{K}$.*

We call a Turing degree **re** iff it contains an re set. Thus both $\mathbf{0}$ and $\mathbf{0}'$ are re degrees, although only the first is "purely re" (meaning that it *only* contains re sets).

7.3 Post's problem for Turing degrees

As was already pointed out, all of the re sets we have encountered hitherto (not counting the construction of theorem 6.7.3) were either recursive or creative. And since creative sets are Turing-complete, it follows that most "common" re sets are either of Turing degree $\mathbf{0}$ (i.e recursive) or $\mathbf{0}'$ (i.e Turing-complete). Thus Post's problem resurfaces in the framework of

Turing-reducibility[4]:

Post's problem: *Is there a non-recursive re set that is not Turing-complete, i.e such that $K \not\leq_T A$? Equivalently, is there an re degree* $\mathbf{a}$ *such that* $\mathbf{0} \prec_T \mathbf{a} \prec_T \mathbf{0}'$*?*

This is a very deep problem whose (affirmative) answer eluded Post and several other researchers for many years.[5] A very involved solution was finally discovered by Friedberg (and, independently, by Muchnik) in 1957, two years after Post's death. Friedberg's pioneer solution was based on a sophisticated argument that later came to be known as the **priority method**. That method has been greatly extended and diversified over the past few decades, spawning a variety of new powerful techniques that have yielded an array of profound results.

Historically, Friedberg's method was anticipated by the work of Post and Kleene, in a result that bears their names. The Post-Kleene theorem was a decisive step towards the positive settlement of the problem: it established the existence of a Turing degree $\mathbf{a}$ that is "in between" $\mathbf{0}$ and $\mathbf{0}'$, i.e such that $\mathbf{0} \prec_T \mathbf{a} \prec_T \mathbf{0}'$. Of course that did not quite solve the problem, as nobody knew whether or not $\mathbf{a}$ is re. Nevertheless, it re-affirmed and lent additional credibility to the widely made conjecture[6] that *there are* re degrees between $\mathbf{0}$ and $\mathbf{0}'$ (it finally turned out there are infinitely many such degrees). And, its ingenious proof paved the way for the priority method.

7.3.1 The Post-Kleene theorem

First we will need a few technical preliminaries.

LEMMA 2.3.1.1. *Re predicates are K-computable.*

Proof. Recall that a predicate $Q \subseteq N^k$ is called re iff there is a Turing-computable predicate $R \subset N^{k+1}$ such that for all $x_1, \ldots, x_k$,

$$Q(x_1, \ldots, x_k) \Longleftrightarrow (\exists y)\ R(x_1, \ldots, x_k, y).$$

What we will actually show is that Q is computable in $KH = \{x \in N \mid \Phi(l(x), r(x)) \downarrow\}$. Since K and KH are recursively isomorphic, $K \equiv_T KH$,

[4] In fact when authors speak of "Post's problem" without qualification, they usually mean the Turing-reducibility version.

[5] Note that the existence of simple sets does not constitute a solution any longer because we have no guarantee that a simple set cannot be Turing-complete. In fact it was later proved (by Dekker) that there are Turing-complete simple sets (actually, his result was even stronger: he proved that *every* re degree —thus including $\mathbf{0}'$— contains a simple set).

[6] Conjectures to the contrary were made as well.

and so $Q \leq_T KH$ will imply $Q \leq_T K$. In particular, let r be the godel number of the Turing machine that computes R, so that

$$\Phi_r(x_1, \ldots, x_k, y) = \begin{cases} 1 & \text{if } R(x_1, \ldots, x_k, y) \\ 0 & \text{if } \neg R(x_1, \ldots, x_k, y). \end{cases}$$

Let $g(x_1, \ldots, x_k) = min_y[\Phi_r(x_1, \ldots, x_k, y) = 1]$. By definition,

$$\begin{aligned} g(x_1, \ldots, x_k) \downarrow &\iff (\exists y)\ [\Phi_r(x_1, \ldots, x_k, y) = 1] \\ &\iff (\exists y)\ R(x_1, \ldots, x_k, y) \\ &\iff Q(x_1, \ldots, x_k). \quad (1) \end{aligned}$$

Define

$$h(x) = \begin{cases} g((x)_1, \ldots, (x)_k) & \text{if } LEN(x) = k \\ \uparrow & \text{otherwise.} \end{cases}$$

Let M_p compute h. Then, with a KH-oracle available, the computation of $Q(x_1, \ldots, x_k)$ is easy: we simply check whether the number $< [x_1, \ldots, x_k], p >$ is in KH. The oracle verdict will be the desired answer, since

$$\begin{aligned} < [x_1, \ldots, x_k], m > \in KH &\iff \Phi_p([x_1, \ldots, x_k]) \downarrow \\ &\iff h([x_1, \ldots, x_k]) \downarrow \\ &\iff g(x_1, \ldots, x_k) \downarrow \\ &\overset{(1)}{\iff} Q(x_1, \ldots, x_k) \downarrow . \end{aligned}$$

Therefore, by the relativized version of Church's thesis, $Q(x_1, \ldots, x_k)$ is KH-computable, and from what we said in the beginning, K-computable as well. ■

LEMMA 2.3.1.2. *If $Q(x_1, \ldots, x_k, y)$ is an re predicate, then the predicate $(\exists y)\ Q(x_1, \ldots, x_k, y)$ is also re.*

Proof. Since Q is re, there is a Turing-computable predicate $R(x_1, \ldots, x_k, y, z)$ such that $Q(x_1, \ldots, x_k, y) \iff (\exists z)\ R(x_1, \ldots, x_k, y, z)$. Thus

$$\begin{aligned} (\exists y)\ Q(x_1, \ldots, x_k, y) &\iff (\exists y)(\exists z)\ R(x_1, \ldots, x_k, y, z) \\ &\iff (\exists t)\ R(x_1, \ldots, x_k, l(t), r(t)), \end{aligned}$$

which shows that re predicates are closed under existential quantification. ■

An **initial segment** of N (or simply "initial segment") is what its name implies: a finite sequence of consequtive numbers beginning with zero, i.e a sequence of the form $\{n\}_{n=0}^{k}$ for some finite $k \in N$. We will use the

letters i, i', i_1, etc., to denote initial segments, and the symbol $\mathbf{I}$ to denote the class of all initial segments. For an initial segment $i = \{n\}_{n=0}^{k}$, we write MAX_i to mean the greatest number in the segment, i.e the number k. Thus if $i = 0, 1, 2, 3$ and $i' = 0, 1, 2, \ldots, 10^{99}$, we have $MAX_i = 3$ and $MAX_{i'} = 10^{99}$.

Now let $\Im$ be the class of all characteristic functions whose domains are initial segments, i.e $\Im = \{f : i \longrightarrow \{0,1\} \mid i \in \mathbf{I}\}$. For $f : i \longrightarrow \{0,1\}$ in $\Im$, define $lh(f)$ (the "length of f") to be $MAX_i + 1$, i.e the cardinality of f's domain (note that $lh(f) = min_x\ [f(x)\uparrow]$). For such a function f, we will write $\Phi^f(x,y)\downarrow$ iff there is a total function $F : N \longrightarrow \{0,1\}$ such that $\Phi^F(x,y)\downarrow$ *and* $f \subset F$ *and* during the course of the computation $\hat{M}_y$ does not access any information on the oracle tape that is to the right of the $lh(f)^{th}$ cell. Intuitively, this means that the finite part f of F provides all the oracle information that is necessary for $\hat{M}_y$ to produce an output when started with input x. We also write $STEP^f(x,y,t)$ iff $\Phi^f(x,y)\downarrow$ in no more than t steps. The three following theorems are fundamental.

THEOREM 2.3.1.1. *The predicate $STEP^f(x,y,t)$ is Turing-computable.*

Proof. We first "write" f on the oracle tape (i.e $f(0)$ on the first cell, $f(1)$ on the second, up to $f(lh(f)-1)$ on the $(lh(f))^{th}$ cell), and then we start executing the program of $\hat{M}_y$ with x as the input. If at some point during the first t steps of the computation $\hat{M}_y$ halts *without* having accessed any oracle cells to the right of the $(lh(f)+1)^{st}$ square, then we output an 1, otherwise we output a zero. By Church's thesis, there exists a Turing machine M that implements this algorithm and thus computes $STEP^f(x,y,t)$. ■

THEOREM 2.3.1.2. *For any total function $G : N \longrightarrow \{0,1\}$,*

$$STEP^G(x,y,t) \Longleftrightarrow (\exists g)\ [(g \in \Im)\ \wedge\ (g \subset G)\ \wedge\ STEP^g(x,y,t)].$$

Proof. The proof is trivial. If some Turing machine $\hat{M}$ with G on the oracle tape halts on input x in no more than t steps, then there is a finite subset $g \subset G$ such that $STEP^g(x,y,t)$. In particular, let the rightmost cell of the oracle tape that was accessed during the computation be the n^{th} cell. Then we simply let

$$g(x) = \begin{cases} G(x) & \text{for } x \leq n \\ \uparrow & \text{otherwise.} \end{cases}$$

Conversely, if such a g exists, then let

$$G(x) = \begin{cases} g(x) & \text{for } x \leq n \\ 0 & \text{otherwise} \quad \text{(an immaterial default value).} \end{cases}$$

Then, clearly, $STEP^g(x, y, t)$. ■

THEOREM 2.3.1.3. *For $x, y \in N$ and $g \in \Im$, the predicate $\Omega(x, y, g) \Longleftrightarrow (\exists f)\ [(f \in \Im) \wedge (g \subset f) \wedge \Phi^f(x, y) \downarrow]$ is K-computable.*

Proof. The trick is to arithmetize the functions in $\Im$. Fix a bijective and algorithmic map $\xi : \Im \longrightarrow N$, and for any $i \in N$ let ξ_i denote the function f in $\Im$ for which $\xi(f) = i$. With this notation, we have, for all x and i, $x \leq lh(\xi_i) \Longrightarrow \xi_i(x) \downarrow$. Thus the predicate $\Omega(x, y, g)$ becomes equivalent to the arithmetic predicate

$$P(x, y, z) \Longleftrightarrow (\exists s)(\exists t)\ [(\xi_z \subset \xi_s) \wedge STEP^{\xi_s}(x, y, t)].$$

Now note that since the condition $\xi_z \subset \xi_s$ is Turing-computable[7] and $STEP^{\xi_s}(x, y, t)$ is Turing-computable (theorem 2.3.1.1), their conjunction is also Turing-computable, and thus the predicate $Q(x, y, z, s) \Longleftrightarrow (\exists t)\ [(\xi_z \subset \xi_s) \wedge STEP^{\xi_s}(x, y, t)]$ is re. Therefore, by lemma 2.3.1.2, the predicate $P(x, y, z) \Longleftrightarrow (\exists s)\ Q(x, y, z, s)$ is re; and, by lemma 2.3.1.1, it is K-computable. And since the original predicate $\Omega(x, y, g)$ is equivalent to $P(x, y, z)$, it follows that it is re. ■

We can now prove the following:

THE KLEENE-POST THEOREM. *There are two incomparable K-computable sets A and B, i.e $A \leq_T K$, $B \leq_T K$, while $A \not\leq_T B$ and $B \not\leq_T A$.*

Proof. We will *construct* A and B so that both are Turing-reducible to K but neither is Turing-reducible to the other. This latter condition means that there is no Turing machine $\hat{M}$ that can either (I) compute c_A with c_B on the oracle tape, or (II) compute c_B with c_A on the oracle tape. Using our notation, these conditions can be rewritten as

$$\left.\begin{array}{ll} (\mathbf{I}') & (\exists x)\ [\ c_A(x) \neq \Phi_y^B(x)\ ] \\ (\mathbf{II}') & (\exists x)\ [\ c_B(x) \neq \Phi_y^A(x)\ ] \end{array}\right\} \text{for } y = 0, 1, 2, 3, \ldots$$

These conditions ensure that for each Turing machine $\hat{M}_y, y \in N$, there exists **(I′)** at least one x such that if $c_A(x) = 1$ (i.e if $x \in A$), then $\Phi_y^B(x) \neq 1$, while if $c_A(x) = 0$ (i.e $x \notin A$), then $\Phi_y^B(x) \neq 0$, so that $A \not\leq_T B$; and, similarly, **(II′)** that there exists at least one x such that $c_B(x) \neq \Phi_y^A(x)$, so that $B \not\leq_T A$.

We will not costruct A and B *per se*, but rather c_A and c_B—which, of course, essentially amounts to the same thing. The construction will proceed

[7] Its computation requires only a finite amount of time.

in stages $s_0, s_1, s_2, \ldots$. At each stage s_n we construct an initial segment of c_A and an initial segment of c_B, that is, two finite characteristic functions a_n and b_n whose domains are initial segments (i.e $a_n, b_n \in \Im$). For each stage s_n, we let i_n and i'_n denote the domains of a_n and b_n respectively, so that

$$(\forall x)\ [x \leq MAX_{i_n} \Longrightarrow a_n(x)\downarrow], \text{ and}$$
$$(\forall x)\ [x \leq MAX_{i'_n} \Longrightarrow b_n(x)\downarrow].$$

Thus for any stage s_n we have

$$a_n = \{(0, a_n(0)), (1, a_n(1)), \cdots, (MAX_{i_n}, a_n(MAX_{i_n}))\}$$
$$b_n = \{(0, b_n(0)), (1, b_n(1)), \cdots, (MAX_{i'_n}, b_n(MAX_{i'_n}))\}.$$

Then at stage s_{n+1} we extend a_n and b_n by defining functions a_{n+1} and b_{n+1} in $\Im$ such that $a_n \subset a_{n+1}$ and $b_n \subset b_{n+1}$. We finally set

$$c_A = \bigcup_{n=0}^{\infty} a_n \quad \text{and} \quad c_B = \bigcup_{n=0}^{\infty} b_n.$$

The stages are obtained as follows: we let y loop through N (beginning with 0) and to each y we correspond *two* stages, one odd (stage s_{2y+1}), and one even (stage s_{2y+2}), as shown in fig.X.

We do that because for *each* oracle Turing machine $\hat{M}_y$ we have to satisfy *two* conditions (**(I′)** and **(II′)**), so we need two stages; at the odd stage (s_{2y+1}) we satisfy **(I′)** and at the even stage (s_{2y+2}) we satisfy **(II′)**. For example, at stage s_3 we make sure that the machine $\hat{M}_{\frac{3-1}{2}} = \hat{M}_1$ does not B-compute A. At stage s_4 we make sure that the same machine ($\hat{M}_{\frac{4-2}{2}} = \hat{M}_1$) does not A-compute B. The details are as follows:

Initializing stage s_0

We let $a_0 = b_0 = \emptyset$, i.e $(\forall x)\ [a_0(x)\uparrow\ \wedge b_0(x)\uparrow]$.

Odd stages s_{2y+1}

We must ensure that the oracle Turing machine $\hat{M}_y$ satisfies condition **(I′)**, i.e that $(\exists x)\ [c_A(x) \neq \Phi_y^B(x)]$. In other words, we must choose a number λ and make sure that $c_A(\lambda) \neq \Phi_y^B(\lambda)$ (A). We choose $\lambda = lh(a_{2y}) = MAX_{i_{2y}} + 1$, i.e the smallest number for which a_{2y} is undefined. We then extend a_{2y} by defining a_{2y+1} to be $a_{2y}(z)$ for $z < \lambda$ and undefined for $z > \lambda$, while for $z = \lambda$ we ensure that $a_{2y+1}(\lambda) \neq \Phi_y^B(\lambda)$ (B). If we satisfy (B) we will also satisfy (A) (and, in turn, **(I′)**), because $a_{2y} \subset c_A$ (by definition). To make matters concrete, suppose that $y = 108$, i.e we are in stage $s_{2\cdot 108+1} = s_{217}$, and that $a_{2y} = a_{216}$ is defined for all $z \leq 374$, i.e $MAX_{i_{216}} = 374$, so that

$$a_{216} = \{(0, a_{216}(0)), (1, a_{216}(1)), \cdots, (374, a_{216}(374))\}.$$

Then $\lambda = lh(a_{216}) = 375$, so we must define a_{217} so that $a_{217}(z) = a_{216}(z)$ for $z < 375$, $a_{217}(z) = \uparrow$ for $z > 375$, and $a_{217}(375) \neq \Phi_{108}^B(375)$ (C). But condition (C) makes sense only if the expression $\Phi_{108}^B(375)$ denotes a number,

and that, in turn, is the case only if $\Phi_{108}^{B}(375)\downarrow$. The problem is that at this point we do not know whether $\hat{M}_{108}$ halts on 375 with B on its oracle tape, for the simple reason that B is an unfinished entity—we have only constructed an initial segment of it, b_{216}. What we do, however, know, is theorem 2.3.1.2, which tells us that *when* B is finished, *then* we will have $\Phi_{108}^{B}(375)\downarrow$ (in a certain number of steps t) *if and only if*

$$(\exists\beta)\ [\beta \in \Im \text{ and } \beta \subset B \text{ and } STEP^{\beta}(375, 108, t)\ ]\ (D).$$

In other words, if the computation is at all to converge, then $\hat{M}_{108}$ will have only used a finite chunk β of oracle information. It is not diffcult to see that if such a β exists in $\Im$, then we must have either $\beta \subset b_{216}$ (E), or $b_{216} \subset \beta$ (F). Now if (E) holds, so does (F) (because if (E) then (D) and (F) are satisfied by $\beta = b_{216}$), so we can say that if (D) holds, then $b_{216} \subset \beta$. Thus we distinguish two cases:

- **Case 1.** *There exists a* $\beta \in \Im$ *such that* $b_{216} \subseteq \beta$ *and* $\Phi_{108}^{\beta}(375)\downarrow$. Then we extend b_{216} to $b_{217} = \beta$, and we define

$$a_{217}(z) = \begin{cases} a_{216}(z) & \text{for } z < 375 \\ \Phi_{108}^{\beta}(375) + 1 & \text{for } z = 375 \ \text{ (the diagonalization step)} \\ \uparrow & \text{for } z > 375. \end{cases}$$

- **Case 2.** *No such* β *exists.* Then theorem 2.3.1.2 implies that $\Phi_{108}^{B}(375)\uparrow$, in which case condition **(I′)** is vacuously satisfied and therefore it does not matter how we define $a_{217}(375)$ or how we extend b_{217}. So we let $b_{217} = b_{216}$ and we define $a_{217}(z)$ to be $a_{216}(z)$ for $z < 375$, undefined for $z > 375$, and zero (a dummy value) for $z = 375$.

To get the general construction read cases 1 and 2 with $lh(a_{2y})$ and y in place of 375 and 108, respectively.

Even stages $\mathbf{s_{2y+2}}$

Repeat the above construction, but with a_{2y+1}, a_{2y} interchanged with b_{2y+2}, b_{2y+1}, respectively.

By this construction, the definition of c_A as $\bigcup_{n=0}^{\infty} a_n$ and c_B as $\bigcup_{n=0}^{\infty} b_n$ ensures that $A \not\leq_T B$ and $B \not\leq_T A$. To see that $A, B \leq_T K$, note that the only uncomputable step in the whole construction was deciding which one of cases 1 and 2 holds. Thus, given any $x \in N$, we could compute $c_A(x)$ and/or $c_B(x)$ if we could decide condition 1. But, by theorem 2.3.1.3, that condition is K-computable, hence c_A and c_B are K-computable, and $A, B \leq_T K$. ■

Appendix A

Why we study partial functions.

The decision to predicate mechanical computability of partial (total *and* non-total) rather than total functions only, is of paramount —and often underrated— theoretical importance. Indeed, if we only admitted total functions there would be very little computability theory and even less *un*computability theory. All computations would be *a priori* known to converge and there would be no halting problem; and no *TOTAL, INFINITE, CO-FINITE*, etc. problems either. But, in the minds of most people, that would actually make things simpler. For, it does seem more straightforward to focus on total functions proper. After all, a cogent argument can be made to show that any general problem can be modeled by a total nt function (chapter 2). So why bother with all the haziness that results from "undefined" values? What follows is an attempt to answer this question.

First of all, some algorithms are actually intended to work and produce results only for a select set of input numbers, their behavior on all other numbers being irrelevant and immaterial. Since the "domain" of such algorithms is a proper subset of N, they are most naturally modeled by non-total functions. We might be interested, for example, in an algorithm that lists the first thousand prime numbers. SInce the behavior of this algorithm is significant only for a finite set of input numbers, we may reasonably say that the algorithm computes the non-total function

$$f(x) = \begin{cases} \text{the } x^{th} \text{ prime number} & \text{if } x \leq 1000 \\ \uparrow & \text{otherwise.} \end{cases}$$

Or we might want to know the x^{th} digit of the decimal expansion of π, but

only when x is a perfect square. An algorithm that solves this problem could be said to compute the function

$$g(x) = \begin{cases} \text{the } x^{th} \text{ digit of } \pi & \text{if } x \text{ is a perfect square} \\ \uparrow & \text{otherwise.} \end{cases}$$

One is likely to think that this is not good enough justification, and that if this is the only purpose they serve, then non-total functions are at best unnecessary and extraneous. After all, why could we not model the problem of listing the first 1000 prime numbers by the *total* function

$$f'(x) = \begin{cases} \text{the } x^{th} \text{ prime number} & \text{if } x \leq 1000 \\ 0 \text{ (or some other default number)} & \text{if } x > 1000. \end{cases}$$

This new function f' is an *extension* of the non-total function f that we defined above. It agrees with f on all numbers on which the latter is defined. And it is certainly mechanically computable, just like f. The only difference is that f' has the added advantage of being total— there is no uncertainty as to whether or where it is defined. So why not dispense with f in favor of f'? Likewise, why not model the second problem by the total function

$$g'(x) = \begin{cases} \text{the } x^{th} \text{ digit of } \pi & \text{if } x \text{ is a perfect square} \\ 10 & \text{otherwise?} \end{cases}$$

More generally, one could remonstrate the endorsement of non-total functions by (erroneously) reasoning as follows: "Non-total functions are not necessary in computability. Give me any non-total Turing-computable function f, and I will turn it into a total Turing-computable function f'. How? Simple: by "patching up" those spots where f is undefined. For instance, I can set

$$f'(x) = \begin{cases} f(x) & \text{if } f(x) \downarrow \\ 7 & \text{if } f(x) \uparrow. \end{cases}$$

The new function f' agrees with f on all arguments on which f is defined. And, since f is computable, f' must be computable too. The only difference is that f' has the advantage of being everywhere defined." The flaw here is in the mistaken assertion "And, since f is computable, f' must be computable too". Let us call a non-total Turing-computable function f **computably extendible** if there is a *total* Turing-computable function f' such that $(\forall x)\ [f(x) \downarrow \Longrightarrow f(x) = f'(x)]$. Then the covert proposition that the foregoing argument really purports to establish is that all non-total

Turing-computable functions are computably extendible. The following theorems proves that this is not true.

THEOREM A1. *There are non-total Turing-computable functions that are not computably extendible.*

Proof. By contradiction. Let $f(x) = \Phi_x(x) + 1$. Since $\Phi_x(x)$ is undefined for some x, f is non-total; and since $\Phi_x(x)$ is Turing-computable, so is f. By assumption, f is computably extendible, so let f' be a total Turing-computable function such that $(\forall x)\ [f(x) \downarrow \Longrightarrow f'(x) = f(x)]$, i.e

$$(\forall x)\ [f(x) \downarrow \Longrightarrow f'(x) = \Phi_x(x) + 1] \quad (1).$$

Since f' is Turing-computable, there is a Turing machine that computes it. Let m be the godel number of that machine. By instantiating equation (1) for $x = m$ we get

$$f(m) \downarrow \Longrightarrow f'(m) = \Phi_m(m) + 1 \quad (2).$$

But the condition $f(m) \downarrow$ is in fact true, because $f(m) = \Phi_m(m) + 1$, and since $\Phi_m = f'$ is total, $\Phi_m(m) = f'(m)$ is some definite number and thus so is $f(m) = \Phi_m(m) + 1$. As a result (by Modus Ponens), (2) yields $f'(m) = \Phi_m(m) + 1$. But $f'(m) = \Phi_m(m)$, thus we derive $\Phi_m(m) = \Phi_m(m) + 1$, a contradiction. ■

However, there is a second, much stronger motivation for treating partial functions. It is the brute fact that *no definition can ever isolate all and only the total functions that are mechanically computable*; diagonalization will always produce a counter-example, that is, a total function that is mechanically computable in the informal sense, but not formally so according the definition (whatever the definition may be). We have already scratched the surface of this in section 2.5, where we maintained that there are no accurate formalizations of 'mechanically computable function' if we take 'function' to mean *total* function. We now expain why.

Assume, as a hypothesis to be proved self-contradictory, that *there is* such an accurate definition; i.e assume that there is a *true* statement D of the form "A total nt function f is mechanically computable iff ...", where the dots are filled with an evidently sharp concept. Two things must be true of such a definition:

(1) The class of functions that it singles out must be countably infinite. For, to each mechanically computable function f there corresponds an algorithm, a method for computing its values— by virtue of which f is mechanically computable. But an algorithm must have a finite verbal description, therefore an uncountable infinity of mechanically computable functions would

entail the existence of an uncountabe infinity of finite strings— an absurdity.
(2) Furthermore, to each function f that it singles out, D must associate at least one formal object o that *computes* f in some way or another, to be determined by the details of D. These "objects" would be intended by the definition D as replacements for the informal and vague entities that we call 'algorithms'. They could be anything: Pascal programs, Turing machines, equational schemas (as in the definition of recursive functions), flowcharts, RAM programs, rewrite systems, anything— exactly what the formal objects would be would depend on the specifics of the definition. But no matter what they are, such objects have finite descriptions in finite languages, and thus they can be uniquely encoded by numbers in an one-one and onto fashion (recall how we encoded entire Turing programs by single numbers). Therefore, there must be an encoding algorithm which will compute the code number of any given formal object o, and a decoding algorithm which will retrieve the object encoded by any given number x. Let o_x be the object encoded by number x in such a scheme, and let ϕ_x bt the total function computed by o_x. Then the list $\phi_0, \phi_1, \phi_2, \ldots$ comprises all and only the total functions that are picked out by D as being mechanically computable. Now define a function $f : N \longrightarrow N$ as

$$f(x) = \phi_x(x) + 1. \quad (A)$$

We observe two things about f:
(i) It is total. And it is total because each ϕ_x is total, which means that the value $\phi_x(x)$ is a definite number; consequently $\phi_x(x) + 1$ is a definite number and f is total.
(ii) It is mechanically computable. To compute $f(x)$ we need only carry out the following algorithm: First retrieve o_x, the "formal object" encoded by x. Then "run o_x on x", whatever that means (the precise meaning of this step would be determined by the specifics of the definition). Since we are assuming that the object o_x computes a total function, i.e that all "computations" halt, an output number r must eventually be obtained. Finally, output the successor of r, i.e the number $r + 1$.

But if f is total and mechanically computable, it must appear somewhere in the list $\phi_0, \phi_1, \phi_2, \ldots$, since that list comprises exactly those functions that are total and mechanically computable. Therefore, there must be a number i such that $f = \phi_i$, so that for all x, $f(x) = \phi_i(x)$ (I). But now consider the number $f(i)$. By definition (equation (A)), $f(i) = \phi_i(i) + 1$ (II), while from (I), $f(i) = \phi_i(i)$ (III). However, (II) and (III) are mutually inconsistent, as

they imply the contradictory proposition $\phi_i(i) = \phi_i(i) + 1$. We conclude, by way of contradiction, that no accurate definition such as D exists.

Bibliography

The following list includes only the works that are cited in this text and those that I have perused in my own studies of the subject. A complete bibliography on computability and uncomputability has been compiled by Hinen and appears on the fourth volume of the Ω-bibliography of Mathematical Logic, Springer Verlag, 1987.

Davis, M.D., and Weyuker, E.J.
[1983] *Computability, Complexity, and Languages*, Academic Press, 1983.

Lewis, H.R., and Papadimitriou, C.H.
[1989] *Elements of the theory of computation*, Prentice-Hall, 1989.

Machtey, M., and Young, P.
[1978] *An introduction to the general theory of algorithms*, North-Holland, 1978.

McNaughton, R.
[1982] *Elementary computability, formal languages, and automata*, Prentice-Hall, 1982.

Minsky M.L.
[1967] *Computation: finite and infinite machines*,
Prentice-Hall, 1967.
Odifreddi, P.
[1989] *Classical recursion theory*,
North-Holland, 1989.
Post, E.L.
[1944] Recursively enumerable sets of positive integers and their decision problems.
Bull. Am. Meth. Soc. 50 (1944) 284–316.
Rogers, H.
[1967] *Theory of recursive functions and effective computability*,
McGraw Hill, 1967.
Salomaa, A.
[1985] *Computation and automata*,
Cambridge University Press, 1985.
Soare, R.I.
[1987] *Recursively enumerable sets and degrees*,
Springer Verlag, 1987.
Tourlakis, G.
[1985] *Computability*,
Reston Publishing Company, 1985.

Also, my opinions on formalizations and definitional accuracy have been considerably influenced by

Goodman, N.
[1966] *The structure of appearance*,
Bobbs-Merrill, 1966.