Elementary

Melvin Fitting
City University of New York

October 8, 2012

1 Introduction

Sometimes problems that seem difficult have simple, but unexpected, solutions. Many, many years ago Raymond acquired a dog, PeekABoo. She was not spayed. The Smullyans lived in the country where dogs often roamed freely, and at a certain point in time we all discovered that their house was surrounded by a ring of attentive male dogs, perhaps a dozen of them. This made it difficult for PeekABoo to use the facilities, so to speak. Now the Smullyan house has a large, enclosed front porch, and this prompted the following successful idea (I no longer remember the collective thought processes that led to it). One person, me I think, closed the door from the porch to the house, and went outside, leaving the outside door open. All the members of the surrounding army rushed into the porch. The outside door was quickly closed. PeekABoo was handed to me through a bedroom window, hurried to my car, and driven a mile away. The dogs were released successfully, PeekABoo was also successful, and it had been demonstrated that this was a problem with a solution. Of course there were other, more permanent, solutions as well, but there was some delay in the application, and I was one of several people who received one of PeekABoo's offspring. My dog was called Benedict, and he was a wonderful companion for many years, through many troubles. But that is another story; this one is about Raymond.

Raymond Smullyan is a man of many parts: musician, puzzle creator of striking originality, mathematical logician. I was (and am) a pupil of Raymond's, and some of my work has involved following up on the consequences of his mathematical ideas. I want to discuss one of them in a non-technical way. The work I will discuss goes back to the late 1950's and early 1960's, and has to do with what computation is. Among other things, his approach has had an influence on the design of programming languages, and in linguistics, but that it another story from the one I will tell.

The year in which I write this is the Turing centenary year, and there are celebrations worldwide. The contribution that Turing is best known for is the creation of the first, and most influential, model of what computation is. A computation is something we recognize when we see one, but that is not the same thing as having a general definition—a general understanding of what it is all about. Turing supplied this. On the practical level, his work influenced the design of the first, and every subsequent, computer. On the theoretical level, Turing's mathematical model of computing machines made it possible to demonstrate the limits of what can be computed. But this is Turing. Where does Raymond come in?

Turing's idealized machines manipulate symbols. This makes sense because, in fact, all our everyday computing is symbol manipulation. For instance our elementary school algorithms for adding integers don't work with numbers at all. Recall, we begin by lining up the right-hand ends of what we are adding. Well, numbers don't have right-hand ends, but *names* for numbers do. All algorithmic human calculation, finally, is symbol manipulation, and Turing built that fact directly into his machines. Nonetheless, we tend to think we are working with numbers, or geometric figures, or whatever. Over 2300 years ago an algorithm was devised (and analyzed) to compute the largest number that divides two integers—the *greatest common divisor*. The algorithm is still very much in use in modern computing. Here is one version. If two integers are different, replace the larger with the result of subtracting the smaller from the larger. Keep doing this until the integers are the same. The final common value is the greatest common divisor. Now if we were to carry this out with pencil and paper a great deal of symbol manipulation would be involved, but if we tried to describe the algorithm in symbol manipulation terms it would be exceedingly hard to understand. We think of it as being about numbers directly, and in fact that's the best way to think about it. What Turing machines can't give us is a direct notion of computability for anything except strings of symbols. For anything else, such as numbers, we need to introduce *names*, and produce algorithms that manipulate names. A more direct approach seems called for. This is the case even though, in practice, we are symbol manipulators. We understand things best when we can talk in the most direct terms. Once we understand what is going on we can translate our understanding into symbol manipulation, but that's not always the right place to start.

In the 1960's and subsequently there was a considerable amount of work by several people, trying to precisely define what computation might mean for things other than symbols (or integers, which are closely related). Many different approaches were developed, some of considerable complexity, but Raymond had already created what may be the simplest of all, in the late 1950's. He called his constructs *Elementary Formal Systems*. The terminology is quite precise. His work is formal, yet what is going on is, in fact, very elementary. Here I will not be formal—instead I will describe things at a more intuitive level. The description is my own, but the ideas are Raymond's.

2 Computing With Boxes

Before we get to the central material, we need to get a small bit of terminology out of the way. We will be computing with—generating—sets and relations. There is a huge area of mathematical research called *set theory*, but we will not need anything of it here except for a few intuitive ideas.

A *set* is just a collection of things—the set of even numbers, the set of people in the world at the present moment, and so on. The primary thing about sets is that either something is a member, or it is not. 4 is in the set of even numbers, 3 is not, for instance, nor is the Atlantic Ocean. Please keep in mind that belonging or not belonging to a set is not the same thing as our knowing whether it does or not.

An example of a relation that most people are familiar with is less-than on integers. Informally, either it 'holds' of a pair of integers, or it does not. $3 < 4$ holds while $7 < 5$ does not. It is convenient for us to make a relation into a kind of set. The less-than relation holds of *pairs* of integers, and order matters. Mathematicians talk about an *ordered* pair—for example, an ordered pair of integers, say $(3, 4)$, which is different than the ordered pair $(4, 3)$. Now, think about the set of all ordered pairs of integers where the first integer is less than the second integer. If we understand the relation $<$, we understand this set. And if we understand this set, we can reconstruct $<$ because, to see if $3 < 4$, look through the set of pairs and see if $(3, 4)$ is there. So, we will follow a common mathematical practice and identify the less-than relation with the set of ordered pairs we just discussed. More generally, a relation on integers is just some set of ordered pairs of integers.

We can (and will) talk about relations of three things, four things, and so on. This means we need ordered triples, quadruples, and so on, but this is straightforward. For example, we will make use of the three-place relation on integers, where the first and second integers add up to the third one. The triples $(2, 3, 5)$ and $(1, 0, 1)$ are in this relation, but the triple $(2, 2, 2)$ is not.

With this out of the way, we can begin our real discussion of Raymond's computation machinery. We begin with computation on the integers 0, 1, 2, $\ldots$, and then say something about other structures. Note that we start with 0 instead of 1, and we do not include negative integers. To keep terminology uncomplicated, we will call these *counting numbers*, though everybody except computer programmers starts counting with 1.

Suppose you are given a (finite) collection of empty boxes, and into these boxes you are to put counting numbers, or ordered pairs of counting numbers, or ordered triples, or $\ldots$. Each box is clearly marked: into this box you can only put counting numbers; into this other box you can only put ordered pairs of counting numbers, and so on. And suppose you are given a list of instructions on how to fill those boxes. The instructions are to be what is customarily called *effective*, which is generally used as an informal term. Some examples of instructions that are *not* effective are: put 7 in box A if today's sunrise was beautiful, or put $(3, 5)$ in box B if the Riemann Hypothesis is true (the Riemann Hypothesis is a famous open mathematical problem). The problem with these instructions is that you (very likely) don't know if you should put the items into the boxes or not. An example of an instruction that *is* effective is: look through box B and if there is an ordered pair there whose first component is, say n, then look through box C and see if there is an ordered triple there whose first component is $n + 1$. If all this is the case, then put $n + 2$ in box A. We might even have unconditional instructions such as: put 7 into box A.

Some general comments. We will only add things to boxes; we will never take things out. We do; we don't undo. This has some direct consequences on what our instructions can be like. We can't have an instruction that says: if 7 is *not* in box A then put $(2, 3)$ in box B. The problem is this. Suppose 7 is not in box A now, so we put $(2, 3)$ in box B. But suppose later on we add 7 to box B. Should we take $(2, 3)$ out of B? Will we have to keep track of the reasons we had for adding each item to a box? What if we could have added something for more than one reason? Horrendous complications arise. It is to avoid such things, and to introduce some stability, that things are only added to boxes. A reason for adding at the present moment remains valid from now on. But then, we can't act on the basis of any negative information, because that might change as we work. In particular, we can't make use of the fact that something is *not* in a box, because later on it might be. So, all our instructions are stated positively: if something *is* in box A, and something else *is* in box B, and so on, add this other thing to this other box.

We have assumed, without calling special attention to it, that we can pick out the first item in an ordered pair, or the second, and also we can form ordered pairs as needed. Similarly for triples, and so on.

There is still one big thing missing. In an example above, we assumed we could do simple arithmetic—given n we could work with $n+1$ and $n+2$. What, exactly, are we allowed to assume? And what is the simplest way of getting this into the picture? Well, the simplest way to get simple arithmetic into the picture is to imagine some of the boxes start off with what we need. These boxes might contain an infinite amount of stuff. This part doesn't concern us—it's just what we are given to start with. For natural numbers, imagine one of the boxes contains the successor relation, that is, one of the boxes contains the ordered pairs $(0, 1), (1, 2), (2, 3), \ldots$. Also imagine another of the boxes contains just the number 0. Informally, we are assuming we can count—we are given where to start, and how to continue. We must take it from here.

And there's one big comment that needs to be made, explicitly and prominently. We are assuming it is numbers we are working with. They are what go in the boxes. Not number names. Not high and low electrical charges. Numbers. Of course, this is impossible. You have never seen a number in a box, though you may have seen the name of a number in a box. We are idealizing here. It's what we have to do in order to describe computation independently from the symbols we must, eventually, use. Imagine we can work with whatever we want, and worry about how later.

Well, that's Raymond's basic set up (though not in his terminology). An Elementary Formal System is a set of boxes, some given already filled, with positive instructions on how to fill the rest. And now we turn to a few examples.

3 Elementary Arithmetic

Our first example is very simple. Assume we have the general setup described in the previous section. In particular we have a box, call it SUC, containing the successor relation, and another box, call it ZERO, containing 0. Let's set up one more box which we'll call POSITIVE (you'll see why in a moment). Here is the rule we'll follow for filling it.

> If a number occurs as the second component of an ordered pair in SUC, put the number in POSITIVE.

How will this work? The numbers we have are 0, 1, 2, ... The number 0 is not the successor of any of these numbers, but all others are successors of something. So, all numbers except 0 can occur as second component of some pair in SUC. So, our rule calls on us to put all the positive integers in POSITIVE.

It will get very wordy if we continue to state our rules as we just did. Instead we introduce variables, used exactly as in high school algebra. They stand for arbitrary numbers. Then, the rule above has the following simpler version.

> If (x, y) is in SUC, put y in POSITIVE.

The next example is a little tricky because it uses something called *recursion*. One of the rules tells us to put something in a box called PLUS provided we have already put something else in that box. We want to fill this box with the addition relation, all triples (x, y, z) where $x + y = z$. The idea is simple. If y is 0, the result of adding $x + y$ is just x. And if we know how to compute, say, $x + 3$, then the result of computing $x + 4$ (where 4 is the successor of 3, of course) is the successor of computing $x + 3$. Now, here are the rules.

> If y is in ZERO then put (x, y, x) in PLUS.
>
> If (x, y, z) is in PLUS and if (y, u) is in SUC and also if (z, v) is in SUC then put (x, u, v) in PLUS.

Following these rules we can, for instance, put $(3, 1, 4)$ into box PLUS. Here's how. To start off, if we take x to be 3 and y to be 0 the first rule tells us: if 0 is in ZERO we can put $(3, 0, 3)$ into box PLUS. Since 0 is in ZERO (the only thing there, in fact), we can put $(3, 0, 3)$ into PLUS. Do so.

Next, taking x to be 3, y to be 0, z to be 3, u to be 1, and v to be 4, the second rule tells us: if $(3, 0, 3)$ is in PLUS, and if $(0, 1)$ is in SUC, and if $(3, 4)$ is in SUC, then put $(3, 1, 4)$ into PLUS. Now both $(0, 1)$ and $(3, 4)$ are in SUC, and we just put $(3, 0, 3)$ into PLUS, so we can put $(3, 1, 4)$ into PLUS.

We can continue like this, and put $(3, 2, 5)$ into PLUS, $(3, 3, 6)$, and so on. In fact, PLUS winds up containing every entry in the addition table. Of course such a table is infinite. What we mean is that by following the instructions, sooner or later we can put any particular entry of it into PLUS. Further, nothing outside the addition table will ever go in.

A word about that word "can." The rules for PLUS tell us what we are *allowed* to do. They do not tell us what we must do. We can put $(3, 1, 4)$ into PLUS, and later on make use of its being there. But we don't have to. What we are interested in is the things that we *could* put into PLUS if we followed the rules zealously. Even then, some of the rules are pretty useless. For instance according to the first rule, if 3 is in ZERO then put $(5, 3, 5)$ in PLUS. This seems strange, but in fact it is something we will never get to use, since 3 is not in ZERO. So, how do we decide what are 'good' values for the variables in our rules? We don't care. There are many systematic ways that can be designed that will allow us to follow the rules without missing anything useful, though useless things may come up along the way. Most of these ways are quite inefficient. All we care about at the moment is what we could do if we always made good choices. We can leave implementation issues to somebody else.

Let's say a relation on counting numbers is *representable* if some set of instructions using some collection of boxes, together with SUC and ZERO, will allow us to put exactly the members of that relation into a box. So, we just showed that the addition relation is representable. In effect, we gave an *Elementary Formal System* for it. Here are a few more examples.

Equality is representable, and this one is really simple. Let EQUAL be a box that holds ordered pairs. Here is the only instruction we need for filling it.

Put (x, x) in EQUAL

However, inequality is more work. We build our way up to it.

The less-than relation is representable. We know that $x < y$ just in case $x + z = y$ for some z that is positive. So, let LESSTHAN be a box that holds ordered pairs, and use the following instruction, combined with instructions and boxes described earlier.

if (x, z, y) is in PLUS and z is in POSITIVE then put (x, y) in LESSTHAN

Finally, combine all of the above, add a box called NOTEQUAL that holds ordered pairs, and use the following instructions.

If (x, y) is in LESSTHAN then put (x, y) in NOTEQUAL

If (y, x) is in LESSTHAN then put (x, y) in NOTEQUAL

You might try showing the algorithm for greatest common divisor, given earlier, can be captured by an Elementary Formal System. That is, give instructions for putting ordered triples (x, y, z) in a box called GCD where: x and y are positive counting numbers and z is their greatest common divisor. This takes some thought, but you have almost everything you will need right now.

4 Besides Arithmetic

The idea is supposed to be that by using boxes the way we did, we can compute with anything, not just with numbers. We need to have some relations given to us to start things off, and then we proceed as above. I'll discuss one example in some detail, and sketch some others. We begin with words, the structure that Turing used.

Suppose we have some finite alphabet, say 0 and 1 to be concrete. (Note that these are symbols now, not numbers.) We can form *words* using this alphabet, 1011 for instance, or even the empty word. The basic operation on words is *concatenation*: word x followed by word y is word z. For instance, the concatenation of 101 and 00 is 10100.

Let's suppose we are given the following boxes to start off with. We have a box CON consisting of all ordered triples (x, y, z) where x concatenated with y is z. We have boxes ZERO, containing the one-letter word 0, and ONE, containing the one-letter word 1. That's it.

The empty word is representable. Suppose we have a box EMPTY that can contain words, and we use the following instruction. (Think about it.)

If (x, y, x) is in CON then put y in EMPTY

Likewise being a non-empty word is representable, as follows. Suppose we have a box NONEMPTY that can contain words, and we use these instructions. (Again, think about it a bit.)

If (x, y, z) is in CON and y is in ZERO then put z in NONEMPTY

If (x, y, z) is in CON and y is in ONE then put z in NONEMPTY

I'll leave it to you to show the shorter-than relation on words is representable.

We might set up machinery for Euclidean geometry, in which we work with points and lines in the plane, and we start off with relations representing when a point is on a line, and so on. We might set up machinery for working with finite sets, or with the binary trees that are common in computer science. All these have, in fact, been investigated. The Elementary Formal System machinery is versatile enough to be applied to anything, as was Raymond's intention.

5 What Have We Got?

Raymond's Elementary Formal Systems are clearly very simple mechanisms, adaptable for use with with pretty much anything. But how does this relate to more familiar notions (at least, notions more familiar to mathematicians)? And where could we go from here?

Let's begin with the machinery introduced above for counting numbers. Here the representable relations are what customarily have been called *recursively enumerable*, though *computably enumerable* is taking its place as terminology. These can be used as the basic building block of the subject called *recursion theory* or *computability theory*, studied both by mathematicians and by theoretical computer scientists.

We noted earlier that Turing machines work directly with words. We set up an Elementary Formal System for computing with words over the alphabet 0 and 1, but any other alphabet could have been used similarly. Now the representable sets are those that are Turing machine acceptable, a standard notion in computer science.

For other structures the representable relations also have nice alternative characterizations. This can be done by introducing names, and using Turing machines, or by using the machinery of mathematical logic. The details are more than we want to go into here. It is enough to point out that of all the approaches designed to capture a plausible notion of direct computing with anything, not just words, Raymond's Elementary Formal Systems are probably the simplest, and the first to be defined, while turning out to be the equivalent of other, more complicated, mechanisms. Many people, myself included, have continued the investigation of Elementary Formal Systems from the point where Raymond left it. As is his habit, Raymond created something elegant that is deep and rich, and yet that looks simple. It is a remarkable habit, and I want to thank him for having it.

Raymond's Works on this Subject

Smullyan, R. M. (1956*a*), "Elementary formal systems (abstract)", *Bulletin of the American Mathematical Society* , Vol. 62, p. 600.

Smullyan, R. M. (1956*b*), "On definability by recursion (abstract)", *Bulletin of the American Mathematical Society* , Vol. 62, p. 601.

Smullyan, R. M. (1961), *Theory of Formal Systems (revised edition)*, Princeton University Press, Princeton, NJ.