Alonzo Church, "An unsolvable problem of elementary number theory"

**Typographical note:**

Right through this section, Church follows Kleene's work in using uses the [Fraktur font](http://en.wikipedia.org/wiki/Fraktur_%28script%29) (boldface) to represent functions that have been defined (and proved recursive) by Kleene. These letters can be found in the [hex Unicode range](http://en.wikipedia.org/wiki/Mathematical_alphanumeric_symbols) of 1D586=𝖆 to 1D59F=𝖟.

Section One: Introduction (pp. 345-346)

This is a fairly brief introduction, so there's not too much to say about it. Since the paper is about problems in *elementary number theory*, Church is trying to motivate his interest in this theory by pointing out that many other branches of mathematics can be reduced to it.

The example he gives in paragraph 2 is Fermat's last theorem. The example in paragraph three is perhaps familiar to mathematicians but is somewhat obscure for computer scientists: you can skim this paragraph.

The concept that all sorts of problems can be reduced to problems involving numbers should not be too strange in CS, since we are used to the idea that a program's inputs and outputs can be represented in *binary format*: as numbers, essentially.

The last paragraph of section 1 (at the top of page 346) is vital, and worth reading carefully for what it tells us about the state-of-the-art when Church wrote this paper. It tells us that:

* Church intends to state precisely what it means to be *effectively calculable*
* In his opinion this concept has not been formalised before now: it's just a "vague intuitive notion"
* Church intends to show that there exists a problem that is well-specified, but not in this class (the "unsolvable" problem in the paper title).

You can read "effectively calculable" as being the equivalent of what we call "computable" - i.e. a problem that we can write a computer program to solve.

The (somewhat long) footnote 2 contains a few important nuggets of information:

* A concept of a *recursive function* has already been formalised by Herbrand and Gödel
* Kleene has been working with this as a definition of "effectively computable"
* Kleene and Church have proved the lambda-calculus and recursive functions to be equivalent: Church believes that this supports the argument that both systems represent "effectively computable"

It's worth emphasising that the last sentence of footnote 2 is **Church's Thesis**: first stated right here.

Section Two: Conversion and lambda definability (pp. 346-349)

In this section, Church introduces and defines the λ-calculus and briefly reviews some results related to the normal form of a formula.

Church doesn't make much of an effort to justify his λ-calculus as a model of computation, and it's not obvious from this paper that the calculus is of much use. He does mention, however, that it's equal in expressive power to Kleene's formalism, so this is the technical justification.

|  |
| --- |
| **pg 346** |

346[7]: Church uses a variety of symbols for parentheses - nowadays we get by with just the usual "(" and ")". Thus, apart from giving a visual indication as to what kind of construct is being parenthesised, you can fairly safely read all these as having the same effect.

346[10]: The first step in defining the λ-calculus is to define its syntax: in the days before formal grammars this is done by the narrative that you see in this paragraph. In particular, a *well-formed formula* is a concept borrowed from logic, and can be seen as meaning a syntactically correct formula.

|  |
| --- |
| **pg 347** |

347[para 2]: Church adopts some conversion here to ease the use of parentheses. It's mildly interesting that his functions, as we expect, take a single parameter at a time, and a function applied to two arguments is initially written along the lines of "(f x) y". The more usual form of "f(x,y)" is introduced as a syntactic abbreviation (no tuples are developed here *within* the formalism) - the kind of conversion that would later become known as [Currying](http://en.wikipedia.org/wiki/Currying).

347[para 3]: Note that the arrow symbol here is not referring to any kind of reduction, but simply a syntactic shorthand. Thus we would expect, when given an formula, to be able to replace all abbreviation names with their formula before starting the reduction process.

347[para 4]: These abbreviations are the [Church numerals](http://en.wikipedia.org/wiki/Church_encoding) - the main idea here is that some number *n* is represented by the application of a function *n* times to an argument. We could justify this choice by developing formulas to represent addition, multiplication etc., but here Church is content simply to have a sequence of formulas that can represent the numbers unambiguously.

347[-11]: These three "operations on well-formed formulas" start with the two main operational rules of the λ-calculus, known nowadays as α and β conversion; rule III is the opposite of β conversion, allowing us to abstract out a free variable using a parameter. Note that a few paragraphs later in 348[3] Church restricts the name *reduction* to mean a conversion that uses only one β conversion and any number of α conversions.

|  |
| --- |
| **pg 348** |

348[5]: Here Church introduces (without much fanfare) one of the most important concepts in the paper: the *normal form* of a formula. The important intuition here is that:

* a λ-calculus formula represents a "program" - i.e. the rules to compute a (mathematical) function
* β-reduction represents the fundamental step of computation
* a normal form represents the *result* of the computation - it's a formula where there's no work left to be done.

348[6]: We usually refer to a part of a formula that has the form {λx[M]}(N) as a *reducible expression*, or a **redex**.

348[para 3]: The concepts of *natural order* and *principal* normal form are, as the footnote says, mainly technical devices to ensure that he can later refer to a normal form unambiguously.

348[para 4]: These theorems are part of the fundamental infrastructure of the λ-calculus. In particular, Theorem II is today known as the**Church-Rosser Theorem**. It is important since, when reducing a formula, there are often a number of possible redexes that you can select. Theorem II says it doesn't matter which one you pick, since *if you get to a formula in normal form*, it will always be the same one. Both Theorems II and II will be needed later in Theorem XVIII on page 361, paragraph 2.

348[last para]: Here Church finally gets round to motivating the calculus that he is presenting: the reduction of a formula to normal form is supposed to represent the calculation of the result of a numeric function.

Note that when referring to mathematical functions, Church uses the term *independent variables* for what we would call the parameters (or arguments), and *dependent variable* for the result. Thus, in the equation *f(x1,...,xn)=y*, each of the *x1...xn* are the independent variables, and *y* is the dependent variable.

|  |
| --- |
| **pg 349** |

349[para 1,2]: Here Church simply points out that numeric functions can be represented in his calculus, without much evidence. This contrast with the typical presentation of the λ-calculus today where we would now start to build up a library of useful functions, as evidence that the calculus was sufficiently powerful. Church's readers would have had to plough through Kleene's long paper (footnote 6) to convince themselves of this.

349[para 3]: Here Church sums up and finally tells us the purpose of section two. This is the payload:

reduction to normal form = effective calculation of a function's value

Section Three: The Gödel representation of a formula (pp. 349-350)

Church has already stated in section one that most problems can be reduced to a format where they become functions over natural numbers. In this section he briefly reviews Gödel numbers, which allow us to express sequences as numbers - this is the key to expressing lambda-terms as numbers, and thus ultimately turning the system upon itself.

349[para 4]: The reference to Gödel numbering in footnote 7 is to Gödel's famous paper *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*, where Gödel used this system to "code up" the workings of predicate logic into operations over natural numbers.

Note how the assignment of numbers to characters in the language is similar to the process of *tokenisation* that happens during the lexical analysis stage of a compiler.

There's a wikipedia article on [Gödel numbering](http://en.wikipedia.org/wiki/G%C3%B6del_numbering), and its basis in maths is the [fundamental theorem of arithmetic](http://en.wikipedia.org/wiki/Fundamental_theorem_of_arithmetic) which tells us that any number can be uniquely factored into products of primes.

349[para 5]: The purpose of the products-of-primes approach is to code up any sequence of numbers into a single number, so that the individual elements can always be factored out afterwards. In language theory terms, if we assign a unique number to each *alphabet symbol*, Gödel numbering allows us then to assign a unique number to each *string*.

As an example, consider the formula we might write as *λab.a(b)*

* This formula is actually *λa[λb[{a} (b)]]* in the original notation.
* The character encoding in paragraph 4 gives us the sequence: 1,17,11,1,19,11,11,17,13,11,19,13,13,13.
* This string (of 14 characters) can then be encoded by listing the first 14 prime numbers, and raising each one to the power of the appropriate character; this comes out as: 21. 317 .511 .71.1119 .1311 .1711 .1917 .2313 .2911 .3119. 3713 .4113 .4313

It's a rather large number, but we can (in principle) calculate it, and, given that number, we can (in principle) break it back down into the product of primes and recover the original number sequence.

The point of all this is that any question about formula is now a question about numbers. For example, consider the problem: *decide if a formula in the λ-calculus is in normal form*.

* Via Gödel numbering, this is equivalent to the problem of deciding if a given number is the *Gödel number* of a formula in normal form.
* Since any number either is or isn't the Gödel number of a formula in normal form, we can think of forming *the set* of all numbers that are Gödel numbers of a formula in normal form.
* Thus any question regarding formulas in the λ-calculus now becomes the problem of deciding membership of a set of natural numbers.

Section Four: Recursive functions (pp. 350-353)

We switch topic here: away from the λ calculus and over to Godel-Herbrand recursive definitions. Note the terminology:

* *functions* are 'real-world' mathematical functions,
* (well-formed) *formulas* are terms written in the λ-calculus
* *expressions* are terms written using the recursive functions notation.

In this section Church defines what it means for a function to be **recursive**. In particular:

* he defines the important concept of a recursive \_function,
* he extends this concept of "recursive" to (infinite) *sequences and properties* (and thus *sets*)
* he extends this further to sequences of *well-formed formulas*

Almost as an aside, he defines the concept of **recursively enumerable** (pg 352, end of paragraph 3).

Since we're working in the realm of natural numbers, the whole paper is thus embedded in the context of **enumerable** sets (those that have a 1:1 correspondence with the natural numbers). A sensible question then is: how do these concepts relate to one another?

By definition, every recursive set is recursively enumerable, and every recursively enumerable set is recursive. It's not explicit here, but the whole paper is essentially an answer to two (related) questions:

* is there a recursively enumerable set that is not recursive?
* is there an enumerable set that is not recursively enumerable?

Theorem XVIII (page 360) will answer the first question in the affirmative, and it first corollary (page 362) will answer the second in the affirmative. Indeed, the point of the whole paper is to formulate and then find an answer to these two questions.

|  |
| --- |
| **pg 350** |

The definition of recursive functions here is perhaps a little strange, particularly since it does not match the usual definition of general recursive functions (via primitive recursion and minimisation). The definition itself was given by Gödel, not as the main definition, but as an *alternative* definition in his 1934 lectures at Princeton.

It is worthwhile to take a quick look at the equations on page 353 to get some idea of the kind of definition Church is trying to describe: systems of equations involving functions applied to variables and other function applications.

350[7]: Note that *1* and *S* (for successor) are the only built-in functions: so natural numbers are hard-wired into this system.

350[8]: When using a function name such as *fn*, the subscript *n* is simply a count of the number of parameters the function takes (called the *arity* of the function).

350[para 5]: Since functions are to be defined by equations, the reasoning here involves either:

* substituting a numeral for one of the variables (all variables are free variables here)
* substituting a term by one that is equal to it

These two kinds of substitution play the same role as *application* and *reduction* played in the λ calculus.

350[last para]: Lots of subscripts and superscripts here! It's worth working your way through these, if only to convince yourself that nothing complicated is going on. Two important points are almost lost in the notation - Church is insisting that, for recursive functions:

* the system is well-defined, so that we can never have *k=l* for different numbers *k* and *l* (thus you can never get two different answers).
* the system can always find the answer: there is always some unique solution (aka "derived equation") for *f(k1,..kn)* for any function *f* and arguments *k1,...,kn*.

|  |
| --- |
| **pg 351** |

351[first para]: In case he seems to be repeating himself: here Church makes the distinction between the function *symbols* that occur in the recursive equations, written using lower-case letters, and the actual functions (real mathematical entities), written using upper case letters. Thus the former "denotes" the latter.

351[third para]: In case we'd missed it, Church makes it explicit that a recursive function is intended to correspond to something that is algorithmically solvable (in finite time) - the conditions that he imposed earlier on being able to always find a unique solution are important here.

351[4th/5th para]: Now Church just notes that since any sequence can be regarded as a function from the natural numbers, we can talk about recursive sequences. Intuitively, a sequence is recursive if we can model it by writing a program that takes some number *i* and generates the ith element of the sequence. We can extend this to properties and relations in the expected way.

|  |
| --- |
| **pg 352** |

352[first para]: Here the concept of *potentially recursive* is what we would recognise as a *partial* function: it's not defined everywhere but, when it is defined, it is equivalent to some recursive function.

352[2nd/3rd para]: The previous section noted that a well-formed formula could be Gödelised as a number, and that operations on formulas could thus be represented by operations on numbers. Here, Church uses this to extend his definition of recursive to formulas, via their Gödel numbers. Since not every operation over formulas is total (e.g. reduction to a normal form), he weakens the requirement on functions over Gödel numbers to be just *potentially* recursive.

352[3rd para]: Notice that Church slips in the definition of recursively enumerable in the very last sentence here (actually, two definitions, given the footnote). At first glance it might be difficult to figure out the difference between a recursive sequence and a recursively enumerable one: essentially Church says that a recursively enumerable set is the *range* of a recursive function. The theme of *potentially* recursive functions is important here, since there's something "partial" about a recursively enumerable set.

Intuitively, given some set *S* and some number *x*,

* *S* is *recursive* if you can answer the question "is *x* in *S*" by writing a program that takes *x* as input and answers "yes" or "no".
* *S* is *recursively enumerable* if you can answer the question "is *x* in *S*" by writing a program that takes *x* as input and answers "yes" if it's in the set, but may answer "no" or fail to terminate if it isn't.

352[last para]: Church builds on the concept of "recursive" to define propositions of elementary number theory (remember the title of the paper) in the expected way - as recursive functions returning 2 for true or 1 for false.

Section Five: Recursiveness of the Kleene 𝖕-function (pp. 353-354)

Church has already defined what he manes by a "recursive" function and, in this section, he takes one of the functions from Kleene's work (called "𝖕") and expresses it as a recursive function - i.e. following the format described in section 4.

This function allows for a kind of iteration: it finds the smallest integer that makes a given function evaluate to true, assuming it exists. There's no particular mystery to its operation - it simply computes the given function with the values 1,2,3,... in succession until it finds one that's true. In Church's formalism, returning "true" is interpreted as returning a value greater than 1.

There are a number of ways of interpreting what this function does, all essentially the same:

* it's a kind of iteration: do the following until a condition becomes true
* it's a kind of search: look through the sequence until you find an element that satisfies this property
* it's a kind of (existential) quantifier: get the least integer *i* that satisfies this property

Church uses this minimisation function extensively in section 8.

|  |
| --- |
| **pg 353** |

**Theorem IV:** The goal here is to produce a set of equations defining a recursive function equivalent to Kleene's 𝖕-function: thus, understanding the theorem really reduces to figuring out what these equations are all about.

Note that in these equations the subscript on the function name just tells you how many arguments that function takes (so he uses "*i2*" because function *i* takes two arguments, not because he wants to define *i1* somewhere).

Looking at the equations as a black-box for a moment: they define a function*f1(x)*which returns the smallest value *y* such that *f2(x,y) > 1*.

The strategy is to compute *f2(x,1)*, *f2(x,2)*, *f2(x,3)*, ... until we find one that has a value >1. To this end, the four internal functions perform the following tasks:

* *h2* initiates the search and selects the answer
* *g2* is the main engine of computation - it's defined recursively (in the modern sense), and it's the one that computes the successive values of *f2*
* *j2* is a predecessor function that returns its first argument minus one; in case the first argument is 1 (and has no predecessor) it returns the second argument.
* *i2* is a comparison function: *g2* uses this to judge if we've found the required value for the current call to *f2*

Each function is defined by cases, so that the arguments are matched against the parameters - no argument can match both 1 and S(x), so this effectively give us the functionality of an if-then-else type of selection. For example, the function *j2* could be written as:

int j2(int x, int y)

{

if (x==1) return y;

else return x-1;

}

Following through the logic of the equations for *f1*, *h2* and *j2* we get:

* *f1* is the entry point, and simply calls *h1* with *1* and its argument
* Note that *h2* is only ever called by *f1*, and only every with the first argument having value 1. Thus we can ignore the first equation of its definition, which I'm assuming is only there for completeness. The second equation for *h2* then applies, effectively telling us to go and compute a value for *g2(x,y)* that has the value 1 (the argument value), and then apply *j2* to this.
* Note that *j2* is only ever called by *h2*, and the first argument is actually the value of *g2(x,y)* and, since the calling point is in the second equation of *h2*, this has the value 1. Thus we only use the first equation in the definition of *j2*, and I'm assuming that the second is only there for completeness.

In summary, given the call to *h2(1,x),* I'm interpreting the definition of *h2* as saying the following: find the value of *g2(x,y)* that returns 1, and then return the value *y*. For this to make sense, *g2(x,y)* should only return 1 for a single value of *y*.

Finally we follow through the logic of *g2* and *i2*:

* Note that the goal here is to get *g2* to return the value 1; since *g2* is defined in terms of *i2*, we're looking for *i2* also to return the value 1. The only equation of *i2* that does this is the second one: hence, when *i2* gets called with its first parameter >1 and the second parameter =2, the process terminates.
* The first equation for *g2* is the base case. It tries out *f2(x,1)* and calls *i2* with the result and the value 2. Thus if *f2(x,1)>1* we return 1 here and the process stops; otherwise *i2* will return 2 and we must continue our search with the second equation for *g2*.
* The second equation for *g2* calculates *f2(x,S( y))* and then calls *i2* with this result and the result from the previous call to *g2*. Based on the definition of *i2*, we only get a 1 returned here if *f2(x,S( y))>1* and the previous call to *g2* returned 2.

Thus I'd interpret the return values of *i2* (in the context of how it's called) as follows:

* return 1 if we've just found the first value of *f2* that's >1
* return 2 if we still haven't yet found a value of *f2* that's >1
* return 3 if we've previously found a value of *f2* that's >1.

For example, if some *f2* has the property that *f2(x,5)* is the smallest value >1, then *i2* will return 2,2,2,2,1,3,3,3,3,...

|  |
| --- |
| **pg 354** |

**Theorem V:** After Theorem IV, the follow-up Theorem V should be easy to work through: this theorem presents a set of recursive equations that finds the *n*th integer such that *F(x)* is true - the last theorem having found the first such integer. The algorithm expressed by these equations is really just two loops; it might be written as:

Input: y, an integer

Input: k, which is the first (smallest) value such that f(k)>1

Output: p, the yth value such that f(p)>1

int i=1;

int p = k;

while ( i < y ) {

// Find the next p such that f(p) holds:

while( f(p)>1 )

p++;

}

return p;

I've had to rename some variables in the above since there were too many uses of x and y; also I've blurred the distinction between an integer and the numeral that represents it.

Actually, the equations really produce a *sequence* of all those values *p* such that *f(p)>1*, starting with the value *k* that was computed in Theorem IV.

* The inner loop is represented by *g2*; note that we keep looping in the first equation for *g2* as long as the first argument is 1, i.e. as long as the current value of *f* is =1. Eventually, when we find a value that is >1 the second equation applies, and we return *y*, the position we've calculated.
* The outer loop is represented by *g1*; this first returns *k*, and then successively returns each value that *g2* finds.

Section Six: Recursiveness of certain functions of formulas (pp. 354-356)

In this short section Church lists a series of results (which he attributes to Kleene) that are essentially results about the operation of the lambda calculus.

Most of the theorem statements have the form: "The set/function/property of .... is recursive/recursively enumerable". It is thus useful to pay attention to those operations deemed *recursive*versus those deemed *recursively enumerable* - Theorems VII, XIV and XV fall into the latter category. It is particularly interesting to ponder the difference between theorems XIII and XIV: at least to convince yourself that the difference between recursive and recursively enumerable requires some care and precision in formulating the problem.

Note that Church has not yet proven that any of the recursively enumerable sets are not also recursive, so one might read "are recursively enumerable" as something like "are *at least*recursively enumerable" - for the moment.

These results will be trooped out in section 8 as scaffolding for the main proof: for example: "By Theorems VI and X there exists a recursive function C ..." (pg. 360[26]).

Section Seven: The notion of effective calculability (pp. 356-358)

This is another section that ties some results together rather than providing more technical apparatus. The summary of the content is given in the last paragraph of the section (page 358): Church wants to link his existing concept of recursiveness and lambda-definability to two others: (i) effective calculability and (ii) provability.

The first part (calculability) is covered in the first three paragraphs of this section, and the second part (provability) is covered in the fourth paragraph.

|  |
| --- |
| **pg 356** |

356[para 3, first half]: this is of interest since Church tries to spell out what might be meant by the intuitive notion of "effective calculability" and then show that this is captured by recursiveness.

He sees an *algorithm* as starting by taking some number *n*, encoding it as some expression, and then deriving subsequent expressions from the previous one, until such time as the process might be regarded as having finished. Presumably each expression along the way represents some "state" of the algorithm, and the answer can then be decoded from the last state.

356[para3, second half]: The function *G(n,x)*takes as input the original argument, *n*, and the current state (actually, history of states) *x*, and calculates the next state in the chain. In the special case where the computation has halted in the current state, the value 10 is returned. (The extra case where *G(n,x)*=1 is just to make the function total).

The function *H(x,n)* simply works like *G* until it terminates, and then reads off (or decodes) the result.

|  |
| --- |
| **pg 357** |

357[main para]: now Church turns to logical systems, and notes that they must have the same general shape as the systems he's been describing to date. In logic, a proof will have some starting point, and each step in the proof should be a valid inference, as controlled by some fixed set of inference rules. Given some fairly harmless assumptions about the recursive enumerability of the process, this is just what we've been dealing with to date.

Thus Church concludes that if we can use logic to define a function, by presenting axioms and rules that say when F(m)=n is provable, then this will be subject to the same constraints as his recursive equations and lambda expressions.

Section Eight: Invariants of conversion (pp. 358-363)

This is the payload of the paper, and the main results are here. It consists roughly of the following:

* **Motivation** (pg. 359): the importance of being able to find what he calls "invariants of conversion" for the transformations in the lambda calculus
* **Lemma** (pg. 359-60): that being able to decide, in general if A converts in to B is "as hard as" being able to decide if C has a normal form
* **Theorem XVIII** (pg. 360-1): the main theorem of the paper: deciding if a formula is normalisable is not recursive
* **Discussion** (pg. 361-2): An informal discussion of Theorem XVIII
* **Corollary 1** (pg. 362): the standard result - if we can find a P that is recursively enumerable (and not recursive) then we can prove that the complement of P is not even recursively enumerable.
* **Corollary 2** (pg. 362-3): just lifts the previous result to a function over integers.
* **Theorem XIX** (pg. 363): just unrolls the Lemma to extend the 'unsolvability' result to the problem of deciding if A converts to B
* **Discussion** (pg 363): Notes that, given the previous discussions, this basically means that the Entscheidungsproblem is undecidable.

|  |
| --- |
| **pg 359** |

**The Lemma:** In this lemma Church shows that the problem of deciding *convertability* is the same as deciding *normalisability*: i.e. if one is recursive then so is the other. This is an "if and only if" kind of statement, so we will need to prove it in both directions: hence the proof is in two parts.

**First part of the Lemma:** If *normalisability* is recursive then so is *convertability*.

Around the middle of page 359, Church uses the rather complicated-looking expression:

{λ xy. 𝖕(λ n. δ(𝖋(x,Z1(n)), 𝖋(y,Z2(n))),1)}   (**a**,**b**)

This is supposed to judge whether the lambda expression represented by **a** is convertible to the lambda expression represented by **b**. It does this by enumerating all the expressions that **a**and **b** can reduce to, and comparing them with each other.

To start picking this apart, note that the formal parameters *x* and *y* match the actual parameters **a** and **b**; so we can plug them in to immediately get:

𝖕(λn. δ(𝖋(**a**,Z1(n)), 𝖋(**b**,Z2(n))), 1)

This is now a call to Kleene's 𝖕 function, which was the subject of section 5.

The 𝖕 function is Kleene's minimisation operator. It takes two arguments - a unary function and a number - and returns:

* 𝖕(D,k) = k, if D(k) converts to 2
* 𝖕(D,k) = 𝖕(D,k+1), if D(k) converts to 1

That is, 𝖕(D,k) finds the smallest value of *i >= k* such that *D(i)=2*.

The value 1 is being supplied to the 𝖕 function above as the place to start counting. Thus the 𝖕 function is trying to find the smallest value of n (starting with 1) for which:

δ(𝖋(**a**,Z1(n)), 𝖋(**b**,Z2(n))) = 2

The function δ is defined in Kleene's work also. It is a *difference* function, so that δ(x,y) is 2 exactly when *x* and *y* are equal (and 1 when they're different).

Thus we are really looking for the smallest value of *n* for which

𝖋(**a**, Z1(n))   =   𝖋(**b**, Z2(n))

Here, *𝖋(x,n)* is a function that assumes an enumeration of all the things that the formula *x* is convertible to, and picks out the nth element of that enumeration. Thus we are comparing the element *Z1(n)* from the enumeration of **a**'s conversions, with the element *Z2(n*) from the enumeration of **b**'s conversions.

Looking at the definitions of *Z1* and *Z2* we see that Church makes use of a function *𝓠*, also borrowed from Kleene's work. Church uses this function when he wants to compare two sequences to each other, element-by-element. Thus he needs to generate the *index set* of the elements he wants to compare. *Z1* is the first index set and *Z2* is the second.

From Kleene's paper we learn that if *R* is some sequence *r1, r2, r3, r4, ...,* then Kleene defines *𝓠(F,R)* to be the sequence:

F(1,1), F(1,2), F(1,3), ....., F(1,r1)  
F(2,1), F(2,2), F(2,3), ....., F(2,r2)  
F(3,1), F(3,2), F(3,3), ....., F(3,r3)  
F(4,1), F(4,2), F(4,3), ....., F(4,r3)  
.....

Church uses this to define his functions *Z1* and *Z2*; in each case he supplies the identity function as the second argument to *𝓠*, so plugging this in we get the sequence *I = 1,2,3,4,5,....,*in place of*R,* so *𝓠(F,I) =*

F(1,1)  
F(2,1), F(2,2),   
F(3,1), F(3,2), F(3,3),   
F(4,1), F(4,2), F(4,3), F(4,4),   
.....

In defining *Z1* he supplies *λx.x(I)* in place of *F*; thus F(x,y) is *(λx.x(I)) y* which is just the identity function applied *x*times to *y* - i.e., just *y*. This yields the sequence *𝓠(λx.x(I), I) =*

1,  
1, 2,  
1, 2, 3,   
1, 2, 3, 4,  
.....

In defining *Z2* he supplies *λxy.S(x)-y* in place of *F* to get the sequence *𝓠(λxy.S(x)-y, I) =*

1,   
2, 1,  
3, 2, 1,  
4, 3, 2, 1,  
.....

Now combine these elements of *Z1* and *Z2* pairwise, and you have an algorithm for enumerating the possible combinations of two sequences - you get the pairs:

(1,1),  
(1,2), (2,1),  
(1,3), (2,2) , (3,1),  
(1,4), (2,3), (3,2), (4,1),  
.....

This then is the last piece of the puzzle: Church is just comparing elements in all pairings of the enumeration of what **a** and **b** convert to, until he finds a pair that are the same.

Note that this formula is only normalisable if we can actually find a pair that are the same. Thus, if we can decide whether or not the formula is normalisable, then we can decide whether or not one formula converts to another.

**Second part of the Lemma:** If *convertability* is recursive then so is *normalisability*.

After the work above, the second formula that Church uses should really be quite straightforward:

{λ x. 𝖕(λ n. 𝖌(𝖋(x,n),1, 1))}   (**c**)

Except: I think there's an extra ",1" in that formula, and it should really read:

{λ x. 𝖕(λ n. 𝖌(𝖋(x,n)), 1)}   (**c**)

We can plug in actual parameter **c** to match the formal parameter x, and see that we are trying to find the minimum value of n (>= 1) in the following:

λ n. 𝖌(𝖋(**c**,n))

This is simply enumerating all the things that **c** converts to, and picking out the first one that is in normal form (since that's what 𝖌 does).

Unfortunately, the result of the 𝖕-function is actually the value of *n* for which this happens, whereas Church insists that he wants the whole thing to reduce to 1. One way of achieving this would be to use the comparison function here (this assumes that the formula isn't already in normal form) and write:

{λ x. δ(𝖕(λ n. 𝖌(𝖋(x,n)), 1) ,1) }   (**c**)

Thus, if **c** is normalisable (in more than one step), the 𝖕-function will return the number of steps, and the δ function will return 1 (i.e. "not equal"), as required. If it isn't normalisable then the 𝖕-function won't return anyway, so we don't have to worry about this. In this way the problem of seeing if a formula has a normal form is equivalent to checking if the overall formula converts to 1.

Aside: using δ(...,1) here is a bit of a hack - I really want to check if something is a number, so a better function might be *λ k. (k (λ x. x) 1)*, which should return 1 for any number. Then we could rephrase the whole formula as:

{λ x. (𝖕(λ n. 𝖌(𝖋(x,n)), 1) (λ x. x) 1) }   (**c**)

|  |
| --- |
| **pg 360-1** |

**Theorem XVIII:** This is the main theorem of the proof: that the problem of deciding if a formula is normalisable is not recursive.

It might be a good idea to read the informal explanation on page 361 (starting "In order to present...") before tackling the theorem itself.

Overall this is a diagonalisation argument: we're going to assume that the property of "having a normal form" is recursive, enumerate all such formulas, and then deliberately build a formula that has a normal form, but isn't in this list.

Church defines a series of functions with their lambda-representations in Fraktur font; these formulas are:

* 𝖍 is a halt-decider: return 2 if the formula halts (i.e. is normalisable), 1 otherwise
* 𝖆 is an enumeration of all the formulas that have a normal form; thus 𝖆(n) is the nth formula that has a normal form.
* 𝖇 is a formula representing "function application"; 𝖇(f,n) is the Gödel-encoding of f(n)
* 𝖈 is the diagonaliser: it will map a result of x to the value x+1 if it is a number, or 1 otherwise
* 𝖉 is just an if-then-else: if the argument is a halt result (=2) then diagonalise, else (=1) just return 1
* 𝖊 welds all this together, to make a normalisable function that's not in the list provided by 𝖆, so giving the contradiction.

Note the definition of 𝖊 is:

𝖊 → λ n. 𝖉(𝖍(𝖇(𝖆(n),𝖟(n))), 𝖇(𝖆(n),𝖟(n)))

There's a common term of "𝖇(𝖆(n),𝖟(n))" in there, so lets' give it a name and tidy up that definition a bit.

We can define:

r → λ n. 𝖇(𝖆(n),𝖟(n))

Thus *r(n)* is just the application of the nth normalisable formula to *n*, (its position in the sequence of such formulas).

Then we can write:

𝖊 → λn. 𝖉(𝖍(r(n)), r(n))

Since *r(n)* is the ingredient for a diagonalisation, the function *𝖊(n)* is going to differ from this *n*th normalisable formula exactly for the argument *n*.

|  |
| --- |
| **pg 362** |

Having proved the main theorem, Church now turns to deriving some straightforward corollaries.

Corollary 1: Since we know that the normalisable formulas are recursively enumerable (Theorem XV) but not recursive (Theorem XVIII), we can deduce that the complement of this set - the non-normalisable formulas - cannot be recursively enumerable.

This is a general result of r.e. sets: if a set and its complement were both r.e. then we could just set the two enumerations going, and we would find any number in one or the other list after a finite amount of time: this the set would in fact be recursive.

Thus, in proving the existence of a set that is r.e. but not recursive, Church has also proven the existence of a set that is not even r.e. (its complement).

Corollary 2: Here Church just re-expresses his non-recursive set (of normalisable formulas) as a function, which is thus also non-recursive.

Note the nice twist here: not only are there some *i* for which *F(i)* is non-calculable, but we cannot even prove it non-calculable (since then we would know beforehand to not bother trying).

Theorem XIX: this is really just a straight consequence: convertability is as hard a normalisability (Lemma pg. 359) and the latter is not recursive, so neither is the former.

Church then concludes that if the Entschiedungsproblem was solvable he would be able to construct some logical statement *Φ(a,b)* saying that *a* converted into *b*, and decide if it was provable, contradicting his result that this is not actually decidable.