A Stronger Foundation for Computer Science and P=NP

Mark Inman, Ph.D.
mark.inman@egs.edu

January 28, 2019

Abstract

This article concerns itself with the relationship between foundational issues in the origins of computer science and how these issues relate to complexity results. We conduct a close reading of the Halting Problem as described by Turing and find a construction of the halting problem where it is fully machine decidable. As the halting problem is **RE**-complete, construction of such a machine implies **RE** = **PSPACE**, which is in direct conflict with known results, implying inconsistency in ZFC. We then search for properties in ZFC that may lead to inconsistency and discover that logicians are using an implied axiom in certain proof by contradiction. We then propose an alternative axiom which will prevent this inconsistency from arising in our future mathematical constructions. With our foundations sound, we are no longer bound by space hierarchy theorems, as such, we then are able to solve the **P** vs. **NP** problem through the construction of a **SPACE-TIME** equivalence oracle.

0 Introduction

The Halting Problem, the consistency of ZFC and the **P** vs. **NP** problem are three foundational concerns that have intrigued computer scientists and mathematicians since their respective conceptions. In particular, the **P** vs. **NP** problem is of great interest because, not only because of it's philosophical implications, but despite being proposed by Stephen Cook nearly a half century ago, there has been no real resolution or any significant advances towards solving this problem. It is mostly assumed that such a problem is still unsolved mainly because solving it will take a major paradigm shift within the field of computer science; a completely new way of thinking about computation.

If we were to ask ourselves why no one has solved the **P** vs. **NP** problem, it makes sense to revisit the very foundations of computer science and mathematical logic. Perhaps one of the reasons this problem has been so elusive is because there is an error in our very mathematical foundations. At the center of the foundations of computer science is Turing's 1936 paper on the Entscheidungsproblem, which proposes a mechanistic way to calculate, store and program numbers. Also in Turing's paper, we find he proposes what is now known as "The Halting Problem" which proves that not all

computable numbers are computably verifiable. So perhaps it is prudent to ask, what if Turing missed something? What if Turing was swept away by the zeitgeist created by Gödel and modernism in general, and his bias was to find only things that couldn't be done, and look past inventive ways which certain problems can be computed?

Understanding this potential bias, we might now ask, what if the assumptions that the number theory from which Turing was working were actually inconsistent? And what if this inconsistency is what leads to contradiction in the Halting Problem? And if it is inconsistency in the axioms for number theory which leads to contradiction, then it is possible that the Turing derived proof that the class of Recursively Enumerable (**RE**) languages are undecidable is actually incorrect.

All we must do to prove this is to provide a non-trivial constructive instance where an **RE**-complete problem is decidable. From deciding an **RE**-complete problem, and finding a counterexample to the Halting problem, we have broken the entire computational complexity hierarchy, and solving for **P** vs. **NP** becomes as easy as setting up an oracle. Prior attempts to set up an oracle to prove **P** = **NP** created contradictions with various hierarchy theorems, but because the **SPACE**-hierarchy theorem depends on an **RE**-complete reduction of the halting problem, if **RE** is decidable, and there exists a constructive counterexample to the Halting problem, then the **SPACE**-hierarchy theorem is invalidated, as their must be a counterexample to the halting problem reduction on which the theorem depends. The existence of a counterexample to an **RE**-complete problem implies not only that the hierarchy proof is wrong, but that the very opposite of the proof is true. We can then begin to build an oracle mechanism which logically solves the **P** vs. **NP** problem.

1 The Halting Problem

1.1 Overview

Turing's monumental 1936 paper "*On Computable Numbers, with an Application to the Entscheidungsproblem*" defined the mechanistic description of computation which directly lead to the development of programmable computers. His motivation was the logic problem known as the Entscheidungsproblem, which asks if there exists an algorithm which can determine if any input of first order logic is valid or invalid. After defining automated computing, he assumed the possibility of a program called an $\mathscr{H}$ Machine which can validate or invalidate its inputs in finite time based on reading a description number for some given program description. Turing proved this $\mathscr{H}$ Machine can not exist through a proof by contradiction. The contradiction appears to arise when $\mathscr{H}$ tries to decide whether or not $\mathscr{H}$ is circle free when given itself as input. But what if we were to program some $\mathscr{H}$ Machine which could recognize when it is deciding for itself? And not through declaring circle-free by fiat, or "rubber stamping", but through an actual process recognition? If such a machine is possible, we have found a valid counterexample to the Halting problem which implies number theory, and by extension, ZFC is inconsistent.

In the following sections, we will discuss the various forms of the halting problem and then give a description on how one would construct a self-validating $\mathscr{H}$ machine if Turing's $\mathscr{D}$ machine were to exist. The proof which determines the inconsistency

of ZFC does not depend on the existence of $\mathscr{H}$, but rather on a reduced version of $\mathscr{H}$, that we will call $\mathscr{J}$. This $\mathscr{J}$ Machine decides only for **RE** problems which are also in **PSPACE** as well as one specific **RE**-complete problem, which is essentially a restricted version of Turing's version of the Halting problem. We will prove that $\mathscr{J}$ exists. The construction of a self-validating $\mathscr{J}$ machine proves **RE** = **PSPACE** which will allow us to prove ZFC (with implied axiom) is inconsistent and to prove that **P** = **NP** through a single oracle call. It also expands our understanding of the theoretical limits of computation and may have application in fault-tolerance of run-time self-correcting code validation in artificial general intelligence implementations. [5]

1.2 The Class of Recursively Enumerable Languages

The class or recursively enumerable languages are specifically those languages which are recognizable by Turing machines. A problem is considered to be in **RE** when a "yes" answer can be verified by a Turing machine in finite time. A problem in **RE** is considered **RE**-complete if any problem in **RE** can be reduced to this problem by a Turing machine, and as such, is the "Hardest" of problems in **RE**.

Turing's version of the Halting problem, described below is the original **RE**-complete problem. All **RE**-complete problems, including the Canonical version of the Halting problem, described later, are reductions from Turing's version of the problem.

1.3 Turing's Version

Turing's version of the halting problem is described completely mechanically without any oracle. It does assume the existence of a machine $\mathscr{D}$ which can decide whether or not a Standard Description of some machine is *circular* or *circle-free* given an arbitrary input. It then posits a machine $\mathscr{H}$ which is a combination of machine $\mathscr{D}$ and a Universal Turing Machine (UTM).

A *Standard Description* or S.D. is the rule set for any given Turing Machine $\mathscr{M}$ in a standard form. By creating a standard, the rule sets themselves can be used to create a *Description Number* or D.N. which itself may be readable by a UTM, $\mathscr{U}$, as an instruction set.

Remark. A Description Number arbitrarily represents a Standard Description. Thus, we can choose which D.N. is used to represent some specific S.D. to our liking, as long as our constructed machine can read the D.N. and interpret it as the corresponding S.D. Again, D.Ns. are arbitrary, thus, if the D.Ns. we receive do not fit our format, where the ordering is consequential to the functioning of $\mathscr{H}_s$, when developing $\mathscr{H}_s$, we can re-assign new D.Ns. (which are not fixed[1]) to the respective S.Ds. (which are fixed) such that $\mathscr{H}_s$ reads the given D.Ns. in the proper order in relationship to c (c, a natural number of relative size defined in section 1.2.1). It is easy to see that Description Numbers are arbitrary for any given Standard Description, because you can always emulate a new machine as a state of the UTM, and have the Description number point to that machine state. Thus, any Standard Description can be represented by

[1]The description numbers are only fixed relative the construction of $\mathscr{H}_s$; we can always re-configure $\mathscr{H}_s$ to change any given D.N. however we wish, as such we regard any D.N. as not fixed.

any Description Number, and the order of solving for a particular Standard Description can be arbitrary for any given solve. This arbitrary solve can be re-ordered to a fixed order if necessary, after the solve is complete.

Let β' be the output 'u' or 's' for all S.D. in a given order as Turing describes below.

From Turing's paper: "Let $\mathscr{D}$ be the Turing Machine which when supplied with the Standard Description (S.D.) of any computing machine $\mathscr{M}$ will test this S.D. and if $\mathscr{M}$ is circular will mark the S.D. with the symbol 'u' and if it is circle free, will mark it with 's' for 'unsatisfactory' and 'satisfactory' respectively. By combining machines $\mathscr{D}$ and $\mathscr{U}$, we could construct a machine $\mathscr{H}$ to compute the sequence of β'"

Turing claims that while $\mathscr{H}$ is circle free by construction, when $\mathscr{H}$ is given the description number for $\mathscr{H}$, it becomes circular. In the eighth section of Turing's paper on the Entscheidungsproblem, Turing claims that the n-th figure of β' is $\phi_n(n)$ can not be determined because of the following reason:

"The instructions for calculating the R(K)-th [figure] would amount to 'calculate the first R(K)-th figures computed by $\mathscr{H}$ and write down the R(K)-th'. This R(K)-th would never be found. I.e. $\mathscr{H}$ is circular..."

This is because when $\mathscr{H}$ tries to evaluate K, it must call its own S.D, which provides instructions on writing outputs from 1 to K-1 in order to call the R(K)-th figure, which tries to evaluate K again, but it can never get there, because it keeps repeating its own instruction loop, writing outputs from 1 to K-1 over and over again as it tries to evaluate for K.

Later, we will construct a *Supermachine* that can recognize itself as its own input from the S.D. It is then programmed to switch to a circle-free state upon this recognition. Because such a construction is arbitrary for any S.D. of this class of Turing Machines, with only relative restrictions on the D.N., we will have found a valid counterexample to Turing's formulation of the halting problem.

Note that this is not "rubber stamping" or programing the machine to declare that it is circle free by fiat. Declaring by fiat means that we pre-feed $\mathscr{H}$ with an instruction to halt upon character recognition of the Description Number. That is considered a trivial counterexample. What we intend to do, by contrast, is to offer a process recognition which can decide for itself as an internally self-recognizing determination, rather than an externally fed recognition state. [5]

1.4 The Canonical Version

1.4.1 Description

The canonical version of the halting problem is easily illustrated by the following pseudocode:

```
def H():
    if halts(h):
        loop_forever()
```

If the subroutine $halts(h)$ halts, h will loop forever, in which case $halts(h)$ is false. Let h be the instructions for $H()$. If $H()$ halts, it will loop forever, which is a contradiction with h, which must halt to be satisfactory, in which case, $H(h)$ does not halt, which means it can not decide h, therefore h is undecidable.

1.4.2 If there is a Counterexample to Turing's Version, then there is a Counterexample to the Canonical Version

The Canonical version of the Halting Problem is actually a Many-one reduction of Turing's Halting Problem whose ancestors are all Many-one reductions to preserve **RE**-completeness (for simplicity of proof, we can just consider this a single Many-one reduction, it applies just the same as when all ancestors are many-one reductions). This version of the halting problem, without a doubt, produces a contradiction. And as a Many-one reduction, may not have a direct counterexample without a call to an oracle. It is because of this reason that we have decided to work with Turing's Halting Problem to find a potential counterexample and not the Canonical Version. Furthermore, if it is discovered that there is a counterexample to Turing's Halting Problem, it is logically valid to call for an oracle from this counterexample to prove a counterexample to any Many-one reductions of Turing's version.

This is easy to prove.

Let P_T be Turing's completely mechanistic description of the Halting problem. Let P_C be the canonical version. Let the operation $\rightarrow_M$ be a many-one reduction function on a problem in **RE**. Let C_T be some counterexample to P_T and C_C be a counterexample to P_C, then we have:

1. Given C_T
2. $C_T \implies P_T$
3. $P_T \rightarrow_M P_C \implies C_T \rightarrow_M C_C$

1.5 A Counterexample to the Undecidability of the Halting Problem

1.6 Self-validating Computers

1.6.1 Supermachines

Let us consider that $\mathscr{H}^{'}$ is a controller machine with a D.N. of $K^{'}$. It controls two different $\mathscr{H}$ machines: $\mathscr{H}_0$ and $\mathscr{H}_1$. $\mathscr{H}_0$ and $\mathscr{H}_1$ each have the ability to determine "u" or "s" on a D.N. input, except $\mathscr{H}_0$ tests as Turing describes, from D.N. 1 counting upwards (Each D.N. is a natural number) and $\mathscr{H}_1$ tests from a certain twos complement of whatever number is being tested by $\mathscr{H}_0$ as a simultaneous parallel input, such that its subsequent D.N. is one less than the previously tested D.N. Let us represent each D.N. by some integer i. $\mathscr{H}_0$ and $\mathscr{H}_1$ have a unique D.N. of K_0 or K_1 respectively.[2]

Upon input of any i_0 to be read by $\mathscr{H}_0$, let $\mathscr{H}^{'}$ store the value pair (i_0, z) until i_0 is determined to be satisfactory or unsatisfactory. When the output is determined, let $\mathscr{H}^{'}$ replace the (i_0, z) with the respective (i_0, s) or (i_0, u) in the data store, such that there is no longer a data store of (i_0, z). Let the same process occur for any i_1, such that $\mathscr{H}^{'}$ also initially stores each D.N. input with (i_1, z) and $\mathscr{H}_1$ reads i_1 to determine satisfactory or unsatisfactory, subsequently replacing the initial value pair with the

[2]This can be determined through a unique identifier string, which does not affect the machine's function or performance, but differentiates the two machines from each other giving them each a unique D.N.

respective value pair (i_1, s) or (i_1, u) depending on the output of $\mathscr{H}_1$. A *redundancy* occurs when some $i_0 = i_1$.

Let $\mathscr{H}'$ have the ability to compare value pairs such that the machine may recognize a redundancy when it occurs, and may also recognize when a value pair contains a z value on the condition of such a redundancy. Let's call this a *z-check* ability.

Let the controller $\mathscr{H}'$ contain a memory command which stores the decision value pairs given by $\mathscr{H}_0$ and $\mathscr{H}_1$

Let $\mathscr{H}_s$ be the supermachine that is the configuration of all three $\mathscr{H}$ Machines as described above and let K_s be the D.N. for the supermachine.

Initialize the identifier strings such that $K_1 < K_0$.

Let the number of bits in $K_0 = n$. Let the twos complement of the first D.N. input to $\mathscr{H}_0$, which is 1, be determined by n such that it satisfies the equation $c = 2^n - 1$.

Initialize K_s to be larger than c.

Lemma. $\mathscr{H}_s$ proceeds circle free, until it reads K_s.

If $c - K_0 > K_1$, then re-initialize the D.N.[3] for either $\mathscr{H}_0$ or $\mathscr{H}_1$ such that $c - K_0 < K_1$. This guarantees that $\mathscr{H}_0$ will read K_1 before $\mathscr{H}_1$ reads K_1 and also guarantees $\mathscr{H}_1$ will read K_0 before $\mathscr{H}_0$ reads K_0. The controller $\mathscr{H}'$ stores the decision value pairs given by $\mathscr{H}_0$ and $\mathscr{H}_1$ through its memory command. The controller may routinely check for a redundancy on the next input.

Now consider when $\mathscr{H}_0$ reads K_1, and K_1 calls the D.N. for $\mathscr{H}_0$: K_0 will call K_1, which will again call K_0 which will result in a z-check, recognizing that the value pair (K_0, z) is already stored in memory, and therefore, since $K_0 < c$, we know that K_0 is the description number for itself, is impossible to call by construction without calling K_1 first, which means it must be checking the description number for a machine which calls itself, namely $\mathscr{H}_1$, which allows us to correctly store the value pair (K_1, s). This same reasoning can be applied for when $\mathscr{H}_1$ reads K_0, correctly storing the value pair (K_0, s).

If however, the machine has determined a redundancy occurred on a value pair where the value is either (i, s) or (i, u) (i.e., a negative evaluation on the z-check, but the redundancy check is positive), then we have already evaluated this D.N. from the other $\mathscr{H}$ machine at the top level, and we no longer have to continue within the range 1 to c, since they will all have been decided. The supermachine, at this point proceeds to utilize machine $\mathscr{H}_0$ and proceeds from D.N. input value $c+1$, and continues through the rest of all Description Numbers, $c + 2$, $c + 3$, etc... at least until it reaches its own D.N., K_s, for no other D.N. need to be problematic for proof[4] as deciding for $\mathscr{H}_s$ and

[3] one may re-initialize, if necessary, the D.N. by adding irrelevant description information into some S.D. yielding a different D.N. provided such information does not affect the integrity of the original S.D.

[4] We should note here the significant finding by Yedidia and Aaronson of the independence of calculating BB(7918) from ZFC which will only halt if and only if ZFC is inconsistent. In a later section of this paper, we prove that a ZFC which allows BB(7918) to be a logically valid construction, is in fact, inconsistent, meaning that BB(7918) is expected to eventually halt. We could thus expect $\mathscr{H}_s$ to determine that BB(7918) will halt. The undecidability of the halting problem, as it is related to BB(7918) is contingent on ZFC with implied axiom being consistent, for BB(7918) will not halt if and only if ZFC is consistent. Forming such a Turing machine, which will halt if and only if the axiom set is inconsistent, using the proposed axiom in the later section of this article, is just not logically sound, as the impredicative form of the machine will

it's constituent parts is RE-complete on their own. Thus, $\mathscr{H}_s$ may proceed circle free, at least until it reaches K_s which is easily constructed to be larger than c. □
[1]

1.7 RE is Decidable

Proof. ***RE** is Decidable.* At the point $K^{'}$ is received as an input, it is determined satisfactory by either $\mathscr{H}_0$ or $\mathscr{H}_1$. Neither K_0 nor K_1 are called during this phase of the process.

By lemma, K_0 is decided by $\mathscr{H}_1$, K_1 is decided by $\mathscr{H}_0$ and $\mathscr{H}_s$ continues indefinitely until we reach K_s, which describes $\mathscr{H}_s$. K_s is read by $\mathscr{H}^{'}$ and as before, its Description Number is stored along with its temporary pair value of z until $\mathscr{H}_0$ or $\mathscr{H}_1$ returns a value for $\beta^{'}$ at that location. K_s is sent to be verified by $\mathscr{H}_0$, which when $\mathscr{H}^{'}$ calls K_s for a second time, under the given recursive property of K_s which will eventually call itself, the z-check for value pair (K_s, z) is recognized as both redundant and with a z value, stored by $\mathscr{H}^{'}$ in the data store, but because the associated value is z, the z-check ability tells us this process has already occurred, sends K_s to $\mathscr{H}_1$, which self-verifies repeated z-check values. By construction, the only value K_i which can provide this multiple z-check values where $K_i > c$ is K_s, so $\mathscr{H}_s$ now self-verifies the input K_s as its own D.N., provides a value of "s" for satisfactory, and halts as a *Circle-free Turing Machine*.

Given that Turing's **RE**-complete halting problem is decidable, it immediately follows that all **RE**-complete problems are decidable by the nature of **RE**-complete being the hardest kind of problem in **RE**.

Let's now construct another **RE**-complete Halting problem, for which the Decider must exist.

1.7.1 Proving a Supermachine Exists

While we have provided a proof that disproves Turing, it may not be immediately clear that such a machine $\mathscr{H}_s$ actually exists. And that even though Turing's proof is invalidated, it is not necessarily true that **RE**=**PSPACE** as we have set out to prove in this section. This is because proving that Turing's proof is incorrect does not automatically prove the existence of the machine we assumed exists, $\mathscr{H}_s$.

But let us not assume this machine exists. Let's construct a machine $\mathscr{J}_s$ which must exist, that is a reduction of $\mathscr{H}_s$, but doesn't assume the existence of $\mathscr{D}$.

It is easy to construct such a machine. Instead of requiring the machine to decide for both **RE** and co-**RE** satisfiability, we can limit the machine to accept only problems in **RE**. This is a perfectly reasonable and valid limitation as our proof only concerns itself with problems in **RE**.

Construct $\mathscr{J}_s$ with the following constituent parts, a decider machine $\mathscr{D}_j$ which runs a program with arbitrary input from beginning to end to determine if the program is circular or circle free. Of course, if ever $\mathscr{J}_s$ takes in a circular machine, $\mathscr{J}_s$ will be circular, but we can prevent this by guaranteeing all inputs to $\mathscr{J}_s$ are in **RE**

violate the proposed axiom. The fact that such a machine can be physically built, is likely a result of the fact consistent systems must be incomplete by Gödel's Second Incompleteness Theorem.

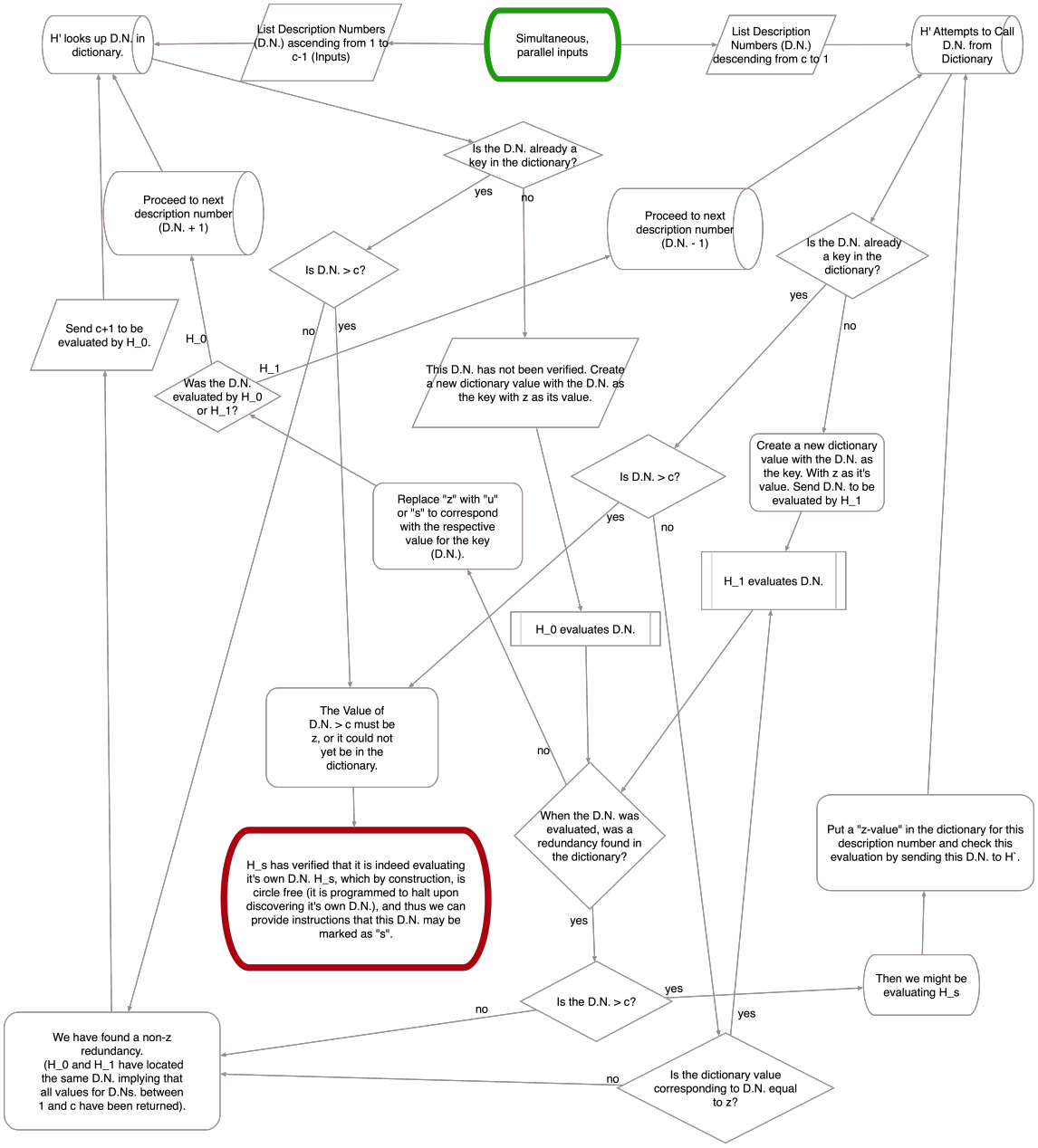

Figure 1: A supermachine configuration appears to exist

with an **RE** constructor, $\mathscr{C}$. This **RE** constructor, is guaranteed to produce a list of, some finite number of, say 2^{1000} D.Ns. for either a deterministic QSAT solver machine $\mathscr{Q}$, and it's respective input of an instance of QSAT, and the D.N. for $\mathscr{J}_s$, inclusive of $\mathscr{J}'$, $\mathscr{J}_0$ and $\mathscr{J}_1$, as we did when we constructed $\mathscr{H}_s$. By including $\mathscr{J}_s$ and its constituent parts as outputs of $\mathscr{C}$, this problem is a many-one reduction of Turing's Halting problem, and is as such, **RE**-complete.

Let these D.Ns. be K_{Qi} for all problems i, for the QSAT solvers, K_{js}, K'_j, K_{j0}, and K_{j1} respectively. $\mathscr{C}$ must initialize $K_{j1} < K_{j0}$. Let n be the number of bits in K_{j0}. Determine c such that $c = 2^n - 1$. $\mathscr{C}$ must order the D.Ns. relative c such that $K_{js} > c$.

We can now re-apply the lemma above for $\mathscr{H}_s$ to $\mathscr{J}_s$, and $\mathscr{J}_s$ halts as a *Circle-free Turing Machine* with **RE**-complete acceptance.

Such a machine is fully constructive and doesn't depend on any assumptions to exist. It is guaranteed to be satisfactory for all QSAT inputs, and the respective **RE**-complete input. The machine is restricted to solving for K_{js} relative c, but if a different output ordering is desired, then a re-sort can be accomplished after solving, making the outputs decidedly enumerable.

Thus, given some *UTM* which can emulate the $\mathscr{J}_s$ Machine, **RE** is decidable.

∎

Corollary: ***RE*=*PSPACE***
It immediately follows that if **RE** is decidable, then **RE**=**PSPACE**.

Even though true, this result is in direct violation of the **SPACE**-hierarchy Theorem. Let us examine the **SPACE**-hierarchy Theorem and see if there is a reason why.

The ***SPACE****-hierarchy Theorem has a counterexample.*

Sipser gives us the following definition: "A function f: $N \to N$, where $f(n)$ is at leaast $O(\log n)$, is called ***space constructable*** if the function that maps the string 1^n to the binary representation of $f(n)$ is computable in space $O(f(n))$."

Sipser continues: "In other words, f is space constructable if some $O(f(n))$ space TM M exists that always halts with the binary representation of $f(n)$ on its tape when started on input 1^n...

"**Space-hierarchy Theorem** For any space constructible function f: $N \to N$, a language A exists that is decidable in $O(f(n))$ space, but not in $o(f(n))$ space...

"The following $O(f(n))$ space algorithm D decides a language A that is not decidable in $o(f(n))$ space.

D= "On input w:

1. Let n be the length of w
2. Compute f(n) using space constructibility, and mark off this much tape. If later stages ever attempt to use more, textitreject.
3. If w is not of the form ¡M¿10* for some TM M, *reject*
4. Simulate M on w while counting the number of steps used in the simulation. If the count ever exceeds $2^{f(n)}$, *reject*
5. If M accepts, *reject*. If M rejects, *accept*."[4]

It is easy to see from this proof that it depends on a reduction of the Halting Problem. So because we have already proven there is a counterexample to the Halting problem, this proof doesn't hold. But that furthermore, it immediately follows that the existence of $\mathscr{J}_s$ implies that there is a machine which decides a language A in both $O(f(n))$ space and $o(f(n))$, and as an immediate consequence of such, **PSPACE=EXPSPACE** □

1.8 Potential Rebuttals to a Counterexample to the Halting Problem

I wish I didn't have to point out that calls to authority, historicity, and other logical fallacies are not proper rebuttals. Making the claim that it is unlikely that a new paper has found a mistake in a proof which is over 80 years old is also not a valid logical argument.

And also, just because many proofs by many very intelligent individuals over the last 80 years depend on RE-complete formulations to prove something can't happen, or that something can't exist, doesn't mean that those proofs are all correct, nor does it imply that these individuals are any less intelligent because of this oversight.

I think it is prudent, given the circumstances, that before we delve further into our final complexity result that $\mathbf{P} = \mathbf{NP}$, that we reflect for a moment on what is really happening here. The results that I have provided in this section are contrary to a plethora of established results, not just the Halting problem and the **SPACE**-hierarchy Theorem. It goes against Kolmogorov Complexity computability bounds, it flies in the face of Rice's theorem, as well as results in set theory yielding higher order large infinite cardinals, and it implies diagonalization is not a sound method for proof. While I directly addressed diagonalization in the first draft of this article, I have since removed it and decided that for simplicity's sake to save it for a followup paper.

What we are now prepared to discover, is that the contradictions which arise from **RE**-complete reductions of the halting problem may arise because of inconsistency in ZFC, not because of proof by contradiction.

2 ZFC with Implied Axiom

2.1 Overview

Finding where the inconsistency occurs in ZFC is not immediately evident from its axiom schema. In fact, what we will find, is that unrestricted use of the axiom of substitution results in a hidden assumption, and that a particularly opinionated stance on this hidden assumption, with proof by contradiction, is being utilized by all mathematicians and logicians today. And that it is with this implied axiom that we find the crux of inconsistency in ZFC. As such, it is not specifically ZFC as it is written that is inconsistent, but rather ZFC with this implied, unwritten axiom that is inconsistent. As such, this is a much lessor result than "ZFC is inconsistent" and doesn't deserve its own paper.

2.2 Incompleteness

Gödel's second incompleteness theorem tells us that any formal system with the expressive power strong enough to represent the proof of its own consistency is either inconsistent or incomplete. This implies that in order to prove the consistency of a formal system such as ZFC, which has the expressive power to represent the proof of its own consistency through the formulation of Peano Postulates together with the Axiom of Substitution, must be incomplete if it is consistent. [3]

We then assume that even though ZFC can express the proof of its own consistency, that it can not determine the proof of its own consistency, and as such, is incomplete, and we say such a proposition is independent of ZFC. However, while true, this is not necessarily the only case. Gödel's second incompleteness theorem gives us a choice. It is possible that ZFC, and similarly expressive formal systems are in fact, also inconsistent. [3]

In order to prove ZFC is inconsistent, we must find two of its theorems which contradict each other. We have already done this in section 1, as **RE** = **PSPACE** contradicts with the **SPACE**-hierarchy theorem. And the existence of the $\mathscr{J}_s$ machine, which solves an **RE**-complete reduction of the Halting problem, contradicts with Turing's proof that **RE** is undecidable.

2.3 Proof by Contradiction

In ZFC, proof by contradiction results from the implication of the law of the excluded middle. When we have all true premises, and an assumption of the existence of a mathematical object, we can prove this object does not exist if the proof leads to contradiction. This is true of course, if and only if ZFC is consistent. If the axiom schema in use is not consistent, then proof by contradiction doesn't necessarily work.

When we examine all the theorems that contradict the findings in section 1 of this paper, we see that all their proofs have certain material similarities. They are all proofs by contradiction. They all employ substitution over free variables which yields to include a *backdoor impredicative*. We can thus say that they are of the *class of proof by contradiction through backdoor impredicative*. We will informally describe, then formally define a backdoor impredicative below.

2.4 Backdoor Impredicatives

An impredicative statement is defined as a statement with self-referencing. Intuitively, we can see how Russell's paradox uses an impredicative definition. Russell's paradox depends on the definition of S as the set of all sets that do not contain themselves. This is impredicative because there is a self-referencing with the definition of the elements of S depending on an element's relationship to itself. The paradox arises as soon as we ask the question whether or not S is a member of itself or not, compounding self reference. Notice that an impredicative definition alone is not enough to create the paradox, this definition must be compounded in some way.

In Russell's paradox, the property of being a set that does not contain itself, is a property applied to a dependent portion of the impredicative statement, that is, the property when applied to sets in general, as a defining element of S, also applies to the

specific set we are defining. This dependence of the defining property of the elements of the contained sets on the containing set, is the distinguishing factor. The point here being that paradoxes of this type may be resolved by removing the compounding impredicative. One way of removing this impredicative, is by introducing the concept of a class. Let a class be a type of set, and thusly, the class of all sets that do not contain themselves, can not contain itself, because classes, by definition cannot contain classes, but only groups sets by type.

But I believe this heuristic falls short. It is not just this dependence that creates a backdoor impredicative statement, it is also the nature of the property itself. We could easily define the set of all sets which contain some element, let's call it x. Such a definition has a dependence on the impredicative portion of the statement, yet does not seem to create a problematic impredicative or an infinite regression or anything of that sort. That is, in order to create a paradox, "x" itself would have to be defined, not only in terms of sets, but in terms of the class of sets in question. As such, in order for impredicatives to be a problem in logic, the property itself must "point back" to the impredicative dependence in a self referential way. Let's call this *impredicative pointing*.

It is this class of impredicative statements which appears problematic, because they seem to contain a logical backdoor which can flip the truth of a theorem, given that there exists some construction of the initial assumption in a proof by contradiction. That more importantly, when there is a backdoor impredicative in a proof by contradiction, the use of this kind of statement in the proof removes the guarantee that there is no counterexample to the theorem.

2.5 Proof by Contradiction through Backdoor Impredicatives

Definition Let *impredicative dependence* be the condition of a statement where a property P depends on self-referencing. $\forall x | P(x) \leftrightarrow \{x \rightarrow P(x)\}$

Definition Let *impredicative pointing* be a condition of self-reference where a property, whose case is dependent on self reference, also references an impredicative dependence, i.e. the existence of S depends on S having an impredicative dependence. $\forall x, \exists S | P(x) \leftrightarrow \{x \rightarrow P(x)\} \rightarrow S$

Definition Let a *backdoor impredicative* be a statement, S which satisfies impredicative pointing with S. $S \vDash \forall x, \exists S | S \leftrightarrow P(S) \leftrightarrow \{\{x \rightarrow P(x)\} \rightarrow S\}$

Consider the following truth table such that:

S	x	P(S)	P(x)	x → P(x)	{x → P(x)} → S	P(S) ↔ {x → P(x)} → S
T	T	T	T	T	T	T
T	T	T	F	F	T	T
T	T	F	T	T	T	F
T	F	T	T	T	T	T
F	T	T	T	T	F	F
T	T	F	F	F	T	F
T	F	T	F	T	T	T
F	T	T	F	F	T	T
T	F	F	T	T	T	F
F	T	F	T	T	F	T
F	F	T	T	T	F	F
T	F	F	F	T	T	F
F	T	F	F	F	T	F
F	F	T	F	T	F	F
F	F	F	T	T	F	T
F	F	F	F	T	F	T

Remark. If the rightmost column is False, the leftmost column can not also be both True and a backdoor impredicative (as the rightmost column is a prerequisite for this kind of statement). Thus, if we assume S is true, if S is provably a backdoor impredicative, the rightmost column must also be True.

Proposition Attempting to disprove the existence of a statement S, with a proof by contradiction through backdoor impredicatives implies disjoint results.

Please note in the truth table above that when we assume S is true, and by substitution we allow, $S = x$, $P(S)$ creates a contradiction by construction when $P(x)$ is false. This is because $P(S)$ must be true when S is true in order for there to be a backdoor impredicative (i.e., the rightmost column is true when S is true). Yet when $P(x)$ is false, x may or may not be false, by rows 2 and 7, setting up the contradiction that $P(x)$ is false when x is false and S is true given the assumption $S = x$. The logical consequence of this contradiction is either $\neg S \oplus \exists x | P(x)$.

However, in an attempt to prove $\neg S$ using a proof by contradiction that relies on a backdoor impredicative, it could be the case that instead of $S = x$, $S \neq x$ and $\exists x$:

1. Given $S \neq x$ by conditional,
2. Given $x \rightarrow P(x)$ by backdoor impredicative,
3. $P(S) \neq P(x)$ by substitution on item 1, implies:
4. $S \neq P(x) \implies \neg S \oplus P(x)$,
5. $\neg S \oplus P(x) \implies \neg S \leftrightarrow \neg P(x)$,
6. $S \leftrightarrow P(x)$, by double negation,
7. $S \leftrightarrow P(x) \implies P(x) \rightarrow S$,
8. $P(x) \rightarrow S$, by items 6, 7,
9. Given x by conditional,
10. $P(x)$ by Modus Ponens on items 2, 9,
11. S by Modus Ponens on item 8.

As such, we can not rely on proof by contradiction through backdoor impredicatative without also assuming that $x = S$. If we can also prove there is some statement $x = \neg S$, then we have not only disproved any theorem that proves $\neg S \& \neg x$, we have proven $\exists x$, whether or not $\exists S | \neg S$. $\square$

Corollary A contradiction in proof by contradiciton through backdoor impredicative may be the result of impredicative pointing, rather than $\neg S$.

When substitution is used to form a backdoor impredicative in order to prove $\neg S$ through proof by contradiction, one must also prove $S \neq x, \nexists\, x | x \rightarrow P(x)$. Current practice is to assume this x does not exist without proof. However, if it can be shown that $\exists x | x \rightarrow P(x) \implies P(x) \rightarrow S$, then this is sufficient to prove the existence of S, even if a contradiction still arises in the proof, as the contradiction does not arise if $\exists S$, the contradiction arises when $\{P(x) \wedge \neg P(S)\} \leftrightarrow \{S = x\}$. Firmly placing contradiction in the hands of impredicative pointing, and not the initial assumption of $\exists S$ for proof by contradiction. $\square$

Corollary $\neg S$ by proof by contradiction through backdoor impredicative iff we assume $\nexists\, x | P(x)$ when $\forall S | P(S)$.

I want to be completely clear here: by examining the second and seventh rows of the truth table, we can see that a contradiction may arise from the use of substitution of S on x to form a backdoor impredicative, which results in confusing $\forall x$ with $\exists S$. However, if $\neg\{x \rightarrow P(x)\}$, then we can not be certain S is false, by truth table rows 2 and 6, because in these cases, both x and S can be true when both $\neg\{x \rightarrow P(x)\}$ and $\neg P(x)$, retaining contradiction for proof when $x = S$. This means that the truth value for S is not determined through proof by contradiction; that is, $P(x)$ can be either true or false for any x and the truth is "hidden". Therefore, the conclusion that $\neg S$ holds iff we assume $\nexists\, x | P(x)$ when $\forall S | P(S)$. $\square$

Let's call such an assumption that $P(x)$ is false (or $\nexists\, x$) when $\neg P(S)$, while also assuming S is true (for proof that $\neg S$ by contradiction), a *hidden assumption.*[5]

2.5.1 The Canonical Halting Problem is a Proof by Contradiction through Backdoor Impredicative

Theorem Turing's proof of the undecidability of the Halting Problem belongs to the class of proofs that are a proof by contradiction through backdoor impredicative.

We may attempt to prove the undecidability of the Halting problem with the following simple version of a halting program:

```
def H():
    if halts(h):
```

[5]As a corollary, S can also be the hidden assumption of a proof by contradiction through backdoor impredicative when the proof openly assumes some property P(x) or x exists in order to disprove either P(x) or x.

```
loop_forever()
```

As has been historically done, we can use this program to prove the undecidability of the Halting problem through a proof by contradiction: If the subroutine $halts(h)$ halts, h will loop forever, in which case $halts(h)$ is false. Let h be the instructions for $H()$. If $H()$ halts, it will loop forever, which is a contradiction with h, which must halt to be satisfactory, in which case, $H(h)$ does not halt, which means it can not decide h, therefore h is undecidable.

To show this proof by contradiction uses a backdoor impredicative, we can designate the function $halts()$ as the property $P()$. We can let the instructions for $H()$, which contains $P()$, be S. We see that the proof lets $h = S$ when it examines β for itself. However, because h is not fixed in all cases, we may designate $h^{'}$ as some arbitrary construction of $H()$, $h^{'} \neq S$.

Proposition. The Halting problem proof contains an impredicative dependence. $\exists h | h \leftrightarrow \{h \rightarrow P(h)\}$

We see this is true, because h contains the instructions for $halts(x)$, which is $P(x)$ therefore $h \leftrightarrow \{h \rightarrow P(h)\}$.

Proposition. The Halting problem proof contains impredicative pointing. $\forall h, \exists S | P(S) \leftrightarrow \{h \rightarrow P(h)\} \rightarrow S$

We can see this is true, because first, $h \in S \implies P(S) \rightarrow \{P(S) \leftrightarrow P(h)\}$. That is S contains h, so any property that applies to S, must also apply to h.

Second, $\exists S, h | \{h \leftrightarrow \{h \rightarrow P(h)\} \rightarrow S\} \rightarrow \{\{h \rightarrow P(h)\} \rightarrow S \implies S \rightarrow P(S)\}$ when $h = S$. That is, if S only exists when there is impredicative dependence, then when S exists in this manner, this implies when we substitute S with h, if S, then $P(S)$.

Third, This is enough to derive that the Halting problem contains impredicative pointing, since $\{\{S \rightarrow P(S)\} \wedge \{h \leftrightarrow \{h \rightarrow P(h)\} \rightarrow S\} \implies P(S) \leftrightarrow P(h) \leftrightarrow \{h \rightarrow P(h)\} \rightarrow S$

Proof. Finally, because the proof contains impredicative pointing, this means that the backdoor impredicaive $S \leftrightarrow H(h) \leftrightarrow \{\{x \rightarrow H(x)\} \rightarrow h\}$ is a logical consequence of the assumptions S and $\neg h^{'}$ through the formulation of the proof by contradiction. Thus, Turing's proof of the undecidability of the Halting problem belongs to the class of proofs that are a proof by contradiction through backdoor impredicative. □

Corollary. $\exists h^{'}$ serves as a counterexample to the proof of the undecidability of the Halting problem.

2.6 The Axiom of Incompleteness

Since all current implementations of ZFC accept hidden assumptions in proofs by contradiction through backdoor impredicative, then the following Axiom is implied by ZFC, even if not explicitly stated, by ZFC.

Axiom $\forall x, \exists S | \{S \leftrightarrow \{P(S) \leftrightarrow \{x \rightarrow P(x)\}\} \rightarrow S\} \rightarrow \forall S, \exists x | \{\neg S \oplus P(x)\} \implies \neg S$

In other words, ZFC implies a particular incompleteness where acceptance of $\nexists x | P(x)$ in the circumstance of a proof of the undecidability of S by contradiction through backdoor impredicative.[6]

[6]The axiom of incompleteness described here is in widespread use by any and all mathematicians and

2.7 ZFC with the Axiom of Incompleteness is Inconsistent

Accepting this axiom of incompleteness, as all logicians of note have since Post, Church and Turing, leads to contradiction when $\exists x|P(x)$. By the existence of $\mathscr{J}_s$ in section 1, the direct implication is that $\exists x|P(x)$, which is in direct contradiction with the above axiom. It immediately follows that ZFC, with the implied axiom above, is inconsistent. □

2.8 Proposition for a New Foundational Axiom

We have demonstrated that unrestricted use of substitution may lead to hidden assumptions in proof by contradiction. The need for a limit on how substitution is applied could help prevent such mistakes from occurring again. Perhaps we could just create a postulate or axiom which makes x a bounded variable after substitution. Such a postulate should allow some impredicative statements, but through preventing certain re-substitutions on x, will prevent impredicative pointing, and thus prevent backdoor impredicatives from forming. We can specify the limit of substitution over bounded x, not to substitution in general, but to statements from the free variable x. Thus, creating a much stronger foundation to our systems of logic and computability.

Axiom. For any formal system Q, with free variables [x; y] and a substitution operation, subst(), z is bounded by subst() such that $\forall x, y, z|subst(x) = y \wedge subst(y) = z \rightarrow subst(z) \neq x$.

3 The P vs. NP Problem

3.1 Overview

The **P** vs. **NP** problem, first described by Stephen Cook in 1972 is the question of whether or not every language accepted by some nondeterministic Turing Machine in polynomial time, is accepted by some deterministic Turing Machine in polynomial time.

Cook informally describes the complexity class **P** as follows: "Informally the class P is the class of decision problems solvable by some algorithm within a number of steps bounded by some fixed polynomial in the length of the input." [2]

The class **NP** can be described informally as the class of decision problems whose answer can be verified in Polynomial time.

3.2 Consequences

It is false to assume that if **P** = **NP** that encryption can or will be broken. It is quite possible that **P** ≠ **PSPACE** and that problems that are **PSPACE**-hard are in exponential time. It is also possible that problems which are linear **SIZE**, such as factoring, are technically in **P**, but still intractable. The same may be said for

computer scientists today.

NP-complete problems like $3SAT$. Anecdotes indicate that many computer scientists have declared that they won't accept any proof that **P** = **NP** without also proof of an algorithm that solves **NP** problems in **P**. I believe this is an unnecessary restriction, especially since it may very well be a fools errand.

However, the possibility of broken encryption is opened by the following proof, just not explicitly the case.

3.3 P=NP

In Section 1, we constructed a Supermachine which is able to solve an **RE**-complete version of the halting problem. In section 2, we tackled the dangers of backdoor impredicatives when applied to proofs by contradiction and proved that ZFC, by accepting hidden assumptions, is inconsistent. In this section, we will expand upon our findings to prove **P** = **NP**.

When proofs by contradiction utilizing a backdoor impredicative are not allowed, we are no longer restricted by them. Furthermore, finding that the halting problem can be solved for all S.D. is in direct contradiction with the **SPACE** hierarchy theorem, because the halting problem is **RE**-complete. However, the **SPACE** hierarchy theorem is a proof by contradiction which utilizes a backdoor impredicative to form a hidden assumption. Since the **SPACE** hierarchy theorem is known to reduce to the Halting problem which has a counterexample, the theorem is invalid by our findings. Similarly, as with other complexity separation arguments, we find that the entire complexity hierarchy collapses, and we now have a foundation in computer science where there is enough information to solve the **P** vs. **NP** problem. Without proof by contradiction with backdoor impredicative, our reasons for not using an oracle to solve **P** vs. **NP** vanish, as the contradictions which formally prevented the use of such oracle no longer exist.

Lemma. If **PSPACE** = **EXPSPACE**, **P** = **NP**.
If the **SPACE** of a problem increases polynomially as with any **PSPACE**-complete problem, this is comparable to the **TIME** of a problem increasing polynomially, such that given an oracle, $=_{Opoly}$, which solves polynomial equivalence between **SPACE** and **TIME**, such that **PSPACE** $=_{Opoly}$ **P**. Similarly, if the **SPACE** of a problem increases exponentially as with any **EXPSPACE**-complete problem, this is comparable to **EXPTIME** which contains **NP**, such that **EXPSPACE** $>=_{Opoly}$ **NP**. If **PSPACE** = **EXPSPACE**, then **PSPACE** $>=_{Opoly}$ **NP**. Since **PSPACE** $=_{Opoly}$ **P**, **P** $>=_{Opoly}$ **NP**, which since **P** and **NP** are both in **TIME**, is the same as **P** = **NP**.

Deciding for the Halting problem in Section 1 is **RE**-complete decidable, and because the **SPACE** hierarchy theorem relies fully on the now defunct method of proof by contradiction utilizing a backdoor impredicative, its results ***must*** be discarded. We may now conclude:

Proof. Since by definition, **PSPACE** $\subseteq$ **RE**, and
since any given Recursively Enumerable set is contained in **PSPACE**,
and $\beta^{'}$ solves for all Recursively Enumerable sets in **PSPACE**,
and since we can no longer accept the **SPACE** hierarchy theorem,
RE $\subseteq$ **PSPACE** ...

RE = **PSPACE**,
such that **EXPSPACE** $\subseteq$ **RE**
and **RE** = **PSPACE**, implies
PSPACE = **EXPSPACE**, proves through the above Lemma ...
P = **NP** ∎

4 Acknowledgments

There are several scientists who I wish to acknowledge, but have each asked me to keep their names in confidence. These scientists, were graduates and/or lecturers at Harvard, Princeton IAS, and MIT respectively. Each tutored me in various capacities in abstract algebra, complexity theory, proof writing and computer science. I would also like to acknowledge a semi-anonymous Slate Star Codex subreddit forum participant by the username of White_Dudeness, who, through our debates on the Halting Problem, allowed me to really hone in on problems with my previous drafts, and to work this revision.

Mark Inman, Ph.D.
Makati City, Metro Manila, Philippines

References

[1] Aaronson, Scott Yedidia, Adam *A Relatively Small Turing Machine Whose Behavior Is Independent of Set Theory* 2016
[2] Cook, Stephen *The P Versus NP Problem* The Clay Mathematics Institute 2000
[3] Gödel, Kurt trans. by Meltzer, B. intro. by Fernau, Christopher *On Formally Undecidable Propositions of Principia Mathematica and Related Systems* 1962
[4] Sipser, M. *Introduction to the Theory of Computation, 3rd ed.*, pp.336-338, Cengage Learning, Boston 2013
[5] Turing, Alan *"On Computable Numbers, with an Application to the Entscheidungsproblem"* 1936-1937: Proceedings of the London Mathematical Society, Series 2, 42, pp 230?265.