A Simple Character String Proof of the “True but Unprovable” Version of Gödel’s First Incompleteness Theorem

Antti Valmari
Tampere University of Technology, Department of Mathematics
PO Box 553, FI-33101 Tampere, FINLAND
Antti.Valmari@tut.fi

A rather easy yet rigorous proof of a version of Gödel’s first incompleteness theorem is presented. The version is “each recursively enumerable theory of natural numbers with 0, 1, $+$, $\cdot$, $=$, $\wedge$, $\neg$, and $\forall$ either proves a false sentence or fails to prove a true sentence”. The proof proceeds by first showing a similar result on theories of finite character strings, and then transporting it to natural numbers, by using them to model strings and their concatenation. Proof systems are expressed via Turing machines that halt if and only if their input string is a theorem. This approach makes it possible to present all but one parts of the proof rather briefly with simple and straightforward constructions. The details require some care, but do not require significant background knowledge. The missing part is the widely known fact that Turing machines can perform complicated computational tasks.

Mathematics Subject Classification 2010: 03F40 Gödel numberings and issues of incompleteness

1 Introduction

Kurt Gödel’s first incompleteness theorem [2] is certainly one of the most important results in mathematical logic. Together with an improvement by Barkley Rosser [11], the theorem says that *any recursive sufficiently strong theory of natural numbers either proves a contradiction, or leaves both some sentence and its negation without a proof.* (We postpone discussion on Gödel’s original formulation to Section 8, because it uses a concept that cannot be explained briefly at this stage. “Recursive” and other background concepts are informally introduced in Section 2.)

More recently, the theorem has often been presented in the form *any recursively enumerable sufficiently expressive theory of natural numbers either proves a sentence that does not hold or fails to prove a sentence that does hold*. This form is not equivalent to Gödel’s and Rosser’s formulation. In some sense it promises less and in some sense more. However, it is easier to prove and perhaps also easier to understand. It makes the assumption of sufficient expressiveness (explained in Section 2) instead of the stronger assumption of sufficient strength (explained in Section 8). This is the version discussed in the major part of this paper. It is compared to Gödel’s and Rosser’s formulation in Section 8.

Both Gödel’s original proof and most of the modern expositions are long and technical. On the other hand, its overall strategy can be explained rather briefly and is intuitively inspiring. As a consequence, the proof is one of the most popularized ones. We only mention here the excellent book by Douglas R. Hofstadter [3]. Unfortunately, to really grasp the result, the technicalities are necessary.

The goal of this paper is to present a *rigorous* proof which, excluding one detail, can be checked *in full* by a reader with *little background* (but not necessarily with little effort). We hope that our proof makes the result accessible to a wider audience than before. The skipped detail is the fact that some simple

Z. Ésik and Z. Fülöp (Eds.): Automata and Formal Languages 2014 (AFL 2014)
EPTCS 151, 2014, pp. 355–369, doi:10.4204/EPTCS.151.25

things can be computed by so-called Turing machines. Its rigorous proof would take many dull pages. On the other hand, Turing machines have been very widely accepted as a universal theoretical model of computation. Therefore, as long as it is obvious that something could be programmed in a modern programming language, it is common practice to skip the proof that a Turing machine can compute it.

Our trick is to first prove that theories of finite character strings with string literals, concatenation, and equality are incomplete. Then we derive the incompleteness of natural number arithmetic as a corollary. In this way, the main constructions of the proof are made using finite character strings, while other proofs make them using natural numbers. This makes our constructions much simpler and much more understandable. The presentation of our proof in this paper is not remarkably short, but this is partly due to the fact that it is very detailed.

Some background concepts are informally explained in Section 2. Our language on finite character strings is defined in Section 3. Not every character can be represented by itself in a string literal. Therefore, an encoding of characters is needed. Section 4 shows that the claim "string y is the sequence of the encodings of the characters in string x" can be formulated in the language. Computations of Turing machines are encoded in Section 5. The incompleteness of theories of strings is shown in Section 6, and of theories of arithmetic in Section 7. Section 8 compares the version of the theorem in this paper to Gödel's and Rosser's versions. Discussion on related work and the conclusions are in Section 9.

An earlier, not peer-reviewed version of this paper appeared as arXiv:1402.7253v1.

2 Informal Background

A *recursive theory* consists of a language for formulating claims about some domain of discourse, together with a recursive proof system. In the case of Gödel's theorem, the domain of discourse is the natural numbers 0, 1, 2, … together with addition ($+$), multiplication (variably denoted with $\times$, $\cdot$, $*$, or nothing such as in $3x+1$), and equality ($=$) with their familiar properties. A *sentence* is a claim without input, formulated in the language. For instance, "3 is a prime number" lacks input but "p is a prime number" has p as input. Of course, whether or not a claim can be formulated depends on the language.

When Gödel published his theorem, the notion of "recursive proof system" had not yet developed into its modern form. Indeed, instead of "recursive", he used a word that is usually translated as "effective". Gödel meant a mathematical reasoning system for proving sentences, where any proof could be checked against a fixed set of straigthforward rules. Proofs were checked by humans, but the requirement was that they could do that in a mechanical fashion, without appealing to intuition on the meaning of formulae. This makes the proof system independent of the different insights that different people might have.

Today, "in a mechanical fashion" means, in essence, "with a computer that has at least as much memory as needed". It suffices that there is a computer program that inputs a finite character string and eventually halts if it is a valid proof, and otherwise runs forever. If such a program exists, then there also is a program that systematically starts the former program on finite character strings one by one in increasing length and executes them in parallel until a proof for the given sentence is found. (We will see in Section 7 how all finite character strings can be scanned systematically.) If the sentence has a proof, the program eventually finds it and halts; otherwise it runs forever in a futile attempt to find a proof.

For mathematical analysis, computers and their programs are usually formalized as *Turing machines*. We will introduce Turing machines in Section 5.

We will not need the assumption that a proof system resembles mathematical reasoning systems. Indeed, we will not need any other assumption than machine-checkability. So we define a *recursively enumerable proof system* as any Turing machine M that reads a finite character string and does or does not halt, such that if the string is not a sentence, then M does not halt. If the string is a sentence, it is considered as proven if and only if M halts. A *recursive proof system* adds to this the requirement of the existence of another Turing machine that halts precisely on those inputs, on which M does not halt.

Section 1 assumed that the language for expressing claims about natural numbers is sufficiently expressive. It suffices that the language has constant symbols 0 and 1, an unbounded supply of variable symbols, binary arithmetic operators $+$ and $\cdot$, binary relation symbol $=$, binary logical operator $\land$ (that is, "and"), unary logical operator $\neg$ (that is, "not"), the so-called universal quantifier $\forall$, and parentheses $($ and $)$. All symbols have their familiar syntactical rules and meanings. The universal quantifier is used to formulate claims of the form $\forall n : P(n)$ (that is, "for every natural number n, $P(n)$ holds").

For convenience, logical or $\lor$ can also be used without changing the expressiveness of the language, because it can be built from $\land$ and $\neg$, since $P \lor Q$ is logically equivalent to $\neg(\neg P \land \neg Q)$. Also logical implication $\rightarrow$, existential quantifier $\exists$, inequality $\neq$, less than $<$, and all familiar numeric constants 2, 3, ..., 9, 10, 11, ... can be used, because $P \rightarrow Q$ is equivalent to $\neg(P \land \neg Q)$, $\exists x : P(x)$ is equivalent to $\neg \forall x : \neg P(x)$, $x \neq y$ is equivalent to $\neg(x = y)$, $x < y$ is equivalent to $(\exists z : x + z + 1 = y)$, and any such numeric constant has the same value as some expression of the form $(1 + 1 + \ldots + 1)$.

3 A First-Order Language on Finite Character Strings

In this section we define our language for expressing claims about *finite character strings*, that is, finite sequences of characters. We use `this font` when writing in that language. To make it explicit where a string in that language ends and ordinary text continues, we put white space on both sides of the string even if normal writing rules of English would tell us not to do so. So we write "characters `a` , ..., `z` are" instead of "characters `a`, ..., `z` are".

The language uses the following characters:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 " \ = ≠ ( ) ~ & | - > A E : + * <
```

We chose this set of characters for convenience. Any finite set containing at least two characters could have been used, at the expense of a somewhat more complicated proof. This set facilitates the use of familiar notation for many things. The fact that its size 53 is a prime number will be exploited in Section 7. The characters `a` , ..., `z` are *lower case letters*, and `0` , ..., `9` are *digits*.

A *finite character string* or just *string* is any finite sequence of characters.

An *encoded character* is `\0` , `\1` , or any other character than `"` and `\` . The encoded character `\0` denotes the character `\` , `\1` denotes `"` , and each remaining encoded character denotes itself. A *string literal* is any sequence of characters of the form `"`α`"` , where α is any finite sequence of encoded characters. It denotes the corresponding sequence of (unencoded) characters. For instance, `""` denotes the empty string and `"backslash=\1\0\1"` denotes the string `backslash="\"` . The purpose of encoding is to facilitate the writing of `"` inside a string literal, without causing confusion with the `"` that marks the end of the literal.

A *variable* is any string that starts with a lower case letter and then consists of zero or more digits. For instance, `a` , `x0` , and `y365` are variables but `49` and `cnt` are not. The value of a variable is a string. We say that the variable *contains* the string.

A *term* is any non-empty finite sequence of variables and/or string literals. It denotes the concatenation of the strings that the variables contain and/or string literals denote. For instance, `"theorem"` , `"theo""rem"` and `"the""""o""rem"` denote the same string `theorem` . If the variable `x` contains the string `or` , then also `"the"x"em"` denotes `theorem` .

An *atomic proposition* is any string of the form t=u or of the form t≠u , where t and u are terms. The first one expresses the claim that the strings denoted by t and u are the same string, and the second one expresses the opposite claim. So `"theorem"="theo""rem"` is a true atomic proposition, and `"theorem"≠"theo""rem"` is not. Indeed, t≠u expresses the same claim as ~t=u , where ~ is introduced soon.

A *formula* is either an atomic proposition or any string of the following forms, where π and ρ are formulae and x is a variable: (π) , ~π , π&ρ , π|ρ , π->ρ , Ax:π , and Ex:π . The parentheses (and) are used like in everyday mathematics, to force the intended interpretation. In the absence of parentheses, formulae are interpreted according to the following precedences: concatenation has the highest precedence, then = , ~ , & , | , -> , and : in this order. For instance, `~b=c|c="hello"&a=bc` denotes the same as `(~(b=c))|((c="hello")&(a=bc))` , and `Ax:~x="8"|x="8"->"u"="u"` denotes the same as `Ax:(((~(x="8"))|(x="8"))->("u"="u"))` . All operators associate to the left, so π->ρ->σ means the same as (π->ρ)->σ .

The formulae express the following claims:

(π) expresses the same claim as π,
~π expresses that π does not hold,
π&ρ expresses that π and ρ hold,
π|ρ expresses that π holds or ρ holds or both hold,
π->ρ expresses that if π holds, then also ρ holds,
Ax:π expresses that for any string x, π holds, and
Ex:π expresses that there is a string x such that π holds.

To improve readability, we often add spaces into a formula, like `Ax: x≠"8" | x="8"` . We may also split a formula onto many lines.

A *first-order language* is any language whose formulae are built from atomic propositions like above. The constants or literals, terms, and atomic propositions of a first-order language may be chosen as appropriate to the domain of discourse. A variable of a first-order language may only contain a value in the domain of discourse, while a variable of a higher-order language may be used more flexibly. When talking about first-order languages in general, we use the symbols ¬, ∀, and so on, and when talking about a particular first-order language specified in this paper, we use ~ , A , and so on.

We will need long formulae. To simplify reading them, we introduce abbreviations. The first abbreviation claims that variable a contains a character, that is, a string of length precisely one. The formula consists of a="x" for each encoded character x, separated by | and surrounded by (and) . We do not write it in full but instead write ··· to denote the missing part.

Char(a) :⇔ (a="a" | a="b" | ··· | a="<")

The abbreviation was written for variable a , but clearly a similar abbreviation can be written for any variable. So we may use the abbreviations Char(b) , Char(g75) , and so on.

The next abbreviation claims that x is a substring of y . That is, there are strings u and v such that string y is the same as string u followed by string x followed by string v .

Sb(x, y) :⇔ (Eu:Ev: y=uxv)

When this abbreviation is used with u in the place of x , some other variable has to be used instead of u on the right hand side. That is, Sb(u, y) does not abbreviate (Eu:Ev: y=uuv) but, for instance, (Ez:Ev: y=zuv) . The incorrect interpretation (Eu:Ev: y=uuv) contains a *name clash*, that is, the variable x that is distinct from u in y=uxv in the definition of Sb(x, y) , became the same variable as u . In general, when interpreting an abbreviation containing a subformula of the form Ax:π

or `E`x`:`π , it may be necessary to replace x by some other variable, to avoid name clashes. Further information on this issue can be found in textbooks on logic, in passages that discuss “bound” and “free” variables.

Please keep in mind that abbreviations are not part of our language. They are only a tool for compactly referring to certain formulae that are too long to be written in full. Each string that uses abbreviations denotes the string that is obtained by replacing the abbreviations by their definitions, changing variable names in the definitions as necessary to avoid name clashes.

4 A Formula Expressing the Encoding of Characters

In this section we show that a formula Q(`x, y`) can be written that claims that `y` is the encoding of `x` , that is, `y` is obtained by replacing `\0` for each `\` and `\1` for each `"` in `x` . We start with a formula claiming that `y` is obtained by replacing `v` for one instance of `u` inside `x` .

```
RepOne(x, u, v, y)  :⇔  ( Ee:Ef: x=euf & y=evf )
```

The next formula claims that, under certain assumptions mentioned below, `y` is obtained by replacing `v` for every instance of `u` inside `x` . It converts `x` to `y` by making the replacements one by one. It assumes that `v` has no characters in common with `u` , so that no fake instances of `u` can occur inside or overlapping `v` . Furthermore, it assumes that different instances of `u` in `x` do not overlap, so that the result is independent of the order in which the instances are chosen for replacement. It also assumes that `p` (for punctuation) is a string that does not occur inside `x` , `y` , or any intermediate result. Furthermore, `p` cannot overlap with itself. We will later see how `p` is constructed.

The sequence of replacements is represented by `s` as a sequence of the form px_1px_2p$\cdots$px_np , where $x_1 =$ x , $x_n =$ y , and $x_2, \ldots, x_{n-1}$ are the intermediate results. The requirements on `p` guarantee that `s` can be decomposed into this form in precisely one way. The parts `(Et: s=pxpt)` and `(Et: s=tpyp)` guarantee that $x_1 =$ x and $x_n =$ y . Thanks to `~Sb(u, y)` , *every* instance of `u` is replaced. The rest of the formula picks each x_i other than the last and claims that x_{i+1} is obtained from it by making one replacement. The x_i is represented by `h` and x_{i+1} by `k` . They are distinguished by not containing `p` , being preceded by `p` , being separated from each other by `p` , and being succeeded by `p` .

```
RepAll(x, u, v, y, p)  :⇔  ( Es:
       (Et: s=pxpt) & (Et: s=tpyp) & ~Sb(u, y)
     & Ah:Ak: ( Sb(phpkp, s) & ~Sb(p, h) & ~Sb(p, k) ) -> RepOne(h, u, v, k)
)
```

To emphasize that abbreviations *are not* but *stand for* strings in our language, and that the strings they stand for are often not easy to comprehend, we now show the string that RepAll(`x, u, v, y, p`) stands for. The real string is too long to be shown on one line, so we split it on two lines.

```
(Es:(Et:s=pxpt)&(Et:s=tpyp)&~(Ez:Ev:y=zuv)&Ah:Ak:((Eu:Ev:s=uphpkpv)&
~(Eu:Ev:h=upv)&~(Eu:Ev:k=upv))->(Ee:Ef:h=euf&k=evf))
```

To obtain the punctuation string `p` , we first make variable `q` contain some sequence of `:`-characters that does not occur inside `x` . Such a string exists, because the string consisting of $n+1$ `:`-characters meets the requirements, when n is the length of `x` .

```
Punct(x, q)  :⇔  ( ( Aa: Sb(a, q) & Char(a) -> a=":" ) & ~Sb(q, x) )
```

The `p` used above is obtained as `"+"q` , that is, by adding a `+`-character to the front of the sequence of `:`-characters in variable `q` . So the value of `p` is `+:::::` or some similar sequence with a different number of `:`-characters. It clearly neither overlaps with itself nor occurs within `x` .

To guarantee that `u` and `v` do not have characters in common, we first convert each instance of `\` to `"*"q` , that is, to some string of the form `*:::···:` that does not occur inside `x` . Then each `*:::···:` is converted to `\0` , then each `"` to `*:::···:` , and finally each `*:::···:` to `\1` . In the first conversion, `u` consists of a single character, so different instances of `u` do not overlap. The same holds for the third conversion. In the second and fourth conversion, `u` is `*:::···:` , which clearly cannot overlap with itself. Furthermore, `+:::···:` does not overlap and is not inside `*:::···:` , so `p` cannot occur in `y` or any intermediate result.

We are now ready to write `Q(x, y)` . In it, the value `\` is represented by the string literal `"\0"` , `\1` by `"\01"` , and so on.

```
Q(x, y)  :⇔  ( Eq:  Punct(x, q)
             & Ex1: RepAll( x, "\0",  "*"q, x1, "+"q)
             & Ex2: RepAll(x1, "*"q, "\00", x2, "+"q)
             & Ex3: RepAll(x2, "\1",  "*"q, x3, "+"q)
             &      RepAll(x3, "*"q, "\01",  y, "+"q) )
```

5 Encoding Turing Machine Computations

Turing machines are a formal model of computation. In this section we show that for each Turing machine, there is a formula `Pvble(x)` that yields true if and only if the machine eventually halts, given `x` as the input. We call it `Pvble(x)` , because the Turing machine is thought to represent some proof system such that it halts if and only if `x` can be proven.

Details of the definition of Turing machines vary in the literature. To start our definition, we introduce a new symbol ⊔, called *blank*. Let *b-strings* be defined similarly to strings, but they may also contain blanks. So our Turing machines use 54 symbols: 53 characters and the blank. A Turing machine consists of a *control unit*, a *read/write head*, and a *tape* that consists of an infinite number of *cells* in both directions. Each cell on the tape may contain any character or ⊔. When we say that some part of the tape is blank, we mean that each cell in it contains ⊔. At any instant of time, the read/write head is on some cell of the tape. During a computation step, the read/write head rewrites the content of the cell and then possibly moves to the previous or the next cell, as dictated by the control unit and the contents of the cell before the step.

The control unit consists of *states* and *rules*. The states are numbered from 0 to r, for some positive integer r. State 0 is called the *final state*. There are $54r$ rules, one for each state q other than 0 and for each character c and ⊔. A rule is of the form $(c,q) \mapsto (c',q',d)$, where c' is any character or ⊔, q' is any state, and d is either `L` , `R` , or `N` . The meaning of the rule is that if the control unit is in state q and the tape cell under the read/write head contains c, then the machine writes c' on the cell, moves the read/write head one cell to the left or right or does not move it, and the control unit enters its state q'. If the control unit enters state 0, then computation halts.

Initially, the tape contains a finite sequence of characters, written somewhere on the tape. This finite sequence is the input to the machine. The rest of the tape is initially blank. Initially, the read/write head is on the first input character (or just anywhere, if the input is empty), and the control unit is in state 1.

At any instant of time, let the *right b-string* mean the content of the cell under the read/write head, the content of the next cell to the right, and so on, up to and including the last character on the tape. If the cell under the read/write head and all cells to the right are blank, then the right b-string is empty. So the last symbol of a non-empty right b-string is always different from $\sqcup$. Let the *left b-string* be defined similarly, but starting at the cell immediately to the left of the read/write head, and proceeding to the left until the first character on the tape is taken. Again, the left b-string may be empty, and if it is not, then its last symbol is not $\sqcup$. The contents of the tape as a whole are an infinite sequence of blanks extending to the left, then the left b-string reversed, then the right b-string, and then an infinite sequence of blanks extending to the right. Initially, the left b-string is empty and the right b-string contains the input.

A halting computation corresponds to a sequence (λ_0, q_0, ρ_0), (λ_1, q_1, ρ_1), $\ldots$, (λ_n, q_n, ρ_n), where λ_0 is the empty string, $q_0 = 1$, ρ_0 is the input string, $q_n = 0$, $q_i \neq 0$ when $0 \leq i < n$, and each (λ_i, q_i, ρ_i) for $1 \leq i \leq n$ is obtained from $(\lambda_{i-1}, q_{i-1}, \rho_{i-1})$ as follows. Here λ_i is the left b-string and ρ_i is the right b-string after i computation steps. Let $c = \sqcup$ if ρ_{i-1} is empty, and otherwise let c be the first symbol of ρ_{i-1}. There is a unique rule of the form $(c, q_{i-1}) \mapsto (c', q', d)$. We have $q_i = q'$. The b-strings λ_i and ρ_i are obtained by replacing c' for the first symbol of ρ_{i-1}, with special treatment of the case that ρ_{i-1} is empty or $c' = \sqcup$; and then possibly moving the first symbol of the resulting b-string to the front of λ_{i-1}, or moving a symbol in the opposite direction, again with some special cases. The special cases are discussed in more detail later in this section. The moving of a symbol from the right b-string to the left b-string models the movement of the read/write head one cell to the right, and the moving of a symbol in the opposite direction models the movement of the read/write head one cell to the left.

We want to model this sequence in our language on strings. The states q_i are represented simply by writing their numbers using the digits `0` , `1` , ..., `9` in the usual way. That is, state number 32768 is represented by `32768` . The b-strings λ_i and ρ_i are more difficult, because they may contain blanks, but there is no blank character in our language. So we represent $\sqcup$ with `\2` and `\` with `\0` . To simplify later constructions by remaining systematic with the encoding in Section 3, we also represent `"` with `\1` . All other characters represent themselves. To summarize, `\` , `"` , and $\sqcup$ on the tape of the Turing machine are represented in λ_i and ρ_i by the values `\0` , `\1` , and `\2` , whose string literal representations are `"\00"` , `"\01"` , and `"\02"` . In this sense, `\` and `"` become doubly encoded.

So we define an *encoded symbol* as `\0` , `\1` , `\2` , or any other character than `\` and `"` . We need not (and could not) say that $\sqcup$ is not an encoded symbol, because $\sqcup$ is not a character at all.

```
EChar(e)  :⇔
     ( e="\00" | e="\01" | e="\02" | Char(e) & e≠"\0" & e≠"\1" )
```

The next formula expresses that `y` is obtained from `x` by replacing `e` for its first encoded symbol, with special treatment of the empty strings and the blank. If `x` consists of at most one encoded symbol, `x` as a whole is overwritten. The result is the empty string if `e` is the encoded blank, and otherwise the result is `e` . If `x` consists of more than one encoded symbols, ordinary replacement occurs.

```
Write(x, e, y)  :⇔  (
        ( x="" | EChar(x) ) & ( e="\02" & y="" | e≠"\02" & y=e )
     | ( Ef:Ez: x=fz & EChar(f) & z≠"" & y=ez )
)
```

Let the encoded form of the left b-string be called *left string*, and similarly with the right b-string. The next formula expresses the removal of the first encoded symbol from one string and its addition to the front of another string, again with special treatment of the empty strings and the blank. The variables `f1` and `f2` contain the values of the from-string before and after the operation, and `t1` and `t2` contain the to-string. The encoded blank `\2` is never added to the front of an empty to-string, to maintain the rule that the b-strings never end with the blank. If the from-string is empty, then the operation behaves as if the encoded blank were extracted from it.

```
Move(f1, t1, f2, t2)   :⇔   (
        ( f1="" & t1="" & f2="" & t2="" )
      | ( f1="" & t1≠"" & f2="" & t2="\02"t1 )
      | ( f1="\02"f2 & t1="" & t2="" )
      | ( Ee: EChar(e) & f1=ef2 & (e≠"\02" | t1≠"") & t2=et1 )
)
```

Next we introduce a formula for each rule $(c,q) \mapsto (c',q',d)$. Let $\dot{c} =$ `\0` , if $c =$ `\` ; $\dot{c} =$ `\1` , if $c =$ `"` ; $\dot{c} =$ `\2` , if $c = \sqcup$; and otherwise $\dot{c} = c$. Let $\ddot{c} =$ `\00` , if $c =$ `\` ; $\ddot{c} =$ `\01` , if $c =$ `"` ; $\ddot{c} =$ `\02` , if $c = \sqcup$; and otherwise $\ddot{c} = c$. We define $\dot{c}'$ and $\ddot{c}'$ similarly. Let $\dot{q}$ denote q written using `0` , `1` , …, `9` in the usual way, and similarly with $\dot{q}'$.

We consider first the case where $d = \mathtt{N}$. The first part of the formula checks that the rule triggers, that is, the current state is q and the symbol under the read/write head is c, taking into accout the possibility that the right string is empty. The second part of the formula gives the state of the control unit, the right string, and the left string new values as dictated by the rule.

```
Rule$_{c,q}^{c',q',\mathtt{N}}$(l1, r1, q1, l2, r2, q2)   :⇔   (
        q1="$\dot{q}$" & ( r1="" & "$\ddot{c}$"="\02" | Ex: r1="$\ddot{c}$"x )
      & q2="$\dot{q}'$" & Write(r1, "$\ddot{c}'$", r2) & l2=l1
)
```

Rules with $d = \mathtt{R}$ or $d = \mathtt{L}$ are similar, but the moving of the read/write head is also represented.

```
Rule$_{c,q}^{c',q',\mathtt{R}}$(l1, r1, q1, l2, r2, q2)   :⇔   (
        q1="$\dot{q}$" & ( r1="" & "$\ddot{c}$"="\02" | Ex: r1="$\ddot{c}$"x )
      & q2="$\dot{q}'$" & Er: Write(r1, "$\ddot{c}'$", r) & Move(r, l1, r2, l2)
)
Rule$_{c,q}^{c',q',\mathtt{L}}$(l1, r1, q1, l2, r2, q2)   :⇔   (
        q1="$\dot{q}$" & ( r1="" & "$\ddot{c}$"="\02" | Ex: r1="$\ddot{c}$"x )
      & q2="$\dot{q}'$" & Er: Write(r1, "$\ddot{c}'$", r) & Move(l1, r, l2, r2)
)
```

Let $\ddot{\lambda}_i$ and $\ddot{\rho}_i$ be obtained from λ_i and ρ_i by replacing each symbol c in them with $\ddot{c}$. The computation of the Turing machine is represented as a string `c` of the form

```
"$\ddot{\lambda}_0$"$\ddot{\rho}_0$\3$\dot{q}_0$"$\ddot{\lambda}_1$"$\ddot{\rho}_1$\3$\dot{q}_1$"$\cdots$"$\ddot{\lambda}_n$"$\ddot{\rho}_n$\3$\dot{q}_n$" .
```

Because the $\dot{q}_i$ consist of just digits and the $\ddot{\lambda}_i$ and $\ddot{\rho}_i$ have been encoded, `"` and `\3` cannot occur inside them. So they can be used for separating the $\ddot{\lambda}_i$, $\ddot{\rho}_i$, and $\dot{q}_i$ from each other.

We are ready to write the formula that claims that the Turing machine halts on input `x` . It says that there is a sequence `c` that models the computation. First, `c` starts with the empty left string, the encoded input string as the right string, and 1 as the state. Second, `c` ends with 0 as the state. Finally, each `"`$\ddot{\lambda}_i$`"`$\ddot{\rho}_i$`\3`$\dot{q}_i$`"`$\ddot{\lambda}_{i+1}$`"`$\ddot{\rho}_{i+1}$`\3`$\dot{q}_{i+1}$`"` satisfies some rule. That `q1` and `q2` do not pick more from `c` than they should follows from the fact that the rules check that they consist of digits only.

```
Pvble(x)   :⇔   ( Ec:
        ( Et:Ey: Q(x, y) & c="\1\1"y"\031\1"t ) & ( Et: c=t"\030\1" )
      & ( Al1:Ar1:Aq1: Al2:Ar2:Aq2:
                ~Sb("\1", l1) & ~Sb("\1", r1) & ~Sb("\1", l2) & ~Sb("\1", r2)
              & Sb("\1"l1"\1"r1"\03"q1"\1"l2"\1"r2"\03"q2"\1", c)
          -> (          Rule$_1$   (l1, r1, q1, l2, r2, q2)
                | $\cdots$ | Rule$_{54r}$ (l1, r1, q1, l2, r2, q2) ) )
)
```

6 Incompleteness of Theories of Finite Character Strings

In this section we prove that any recursively enumerable proof system for our language on strings either fails to prove some true sentence, or proves some false sentence.

Please remember that Q(x, y) and Pvble(x"\1"y"\1") are abbreviations used in this paper to improve readability, and not as such strings in our language. They stand for some strings in our language that are too long to be written explicitly in this paper. Each of these two long strings has a corresponding encoded string, which is obtained by replacing \0 for each \ and \1 for each " . We denote them with Q̇(x, y) and Ṗvble(x"\1"y"\1") . Also remember that Q(x, y) claims that y is the encoded form of x . Therefore,

Q(Q(x, y), Q̇(x, y)) and Q(Pvble(x"\1"y"\1"), Ṗvble(x"\1"y"\1")) hold.

We can now write Gödel's famous self-referential sentence in our framework as follows.

```
 Ex:Ey: Q(x, y) & ~Pvble(x"\1"y"\1") & x=
"Ex:Ey: Q̇(x, y) & ~Ṗvble(x"\1"y"\1") & x="
```

Let α be any string and β be its encoded form. Then x="β" says that the variable x has the value α. Therefore, the last part of Gödel's sentence says that the value of x is the following string, with Q(x, y) and Pvble(x"\1"y"\1") replaced by the strings they stand for:

```
Ex:Ey: Q(x, y) & ~Pvble(x"\1"y"\1") & x=
```

This and Q(x, y) together say that the value of y is the following string, with Q̇(x, y) and Ṗvble(x"\1"y"\1") replaced by the strings they stand for:

```
Ex:Ey: Q̇(x, y) & ~Ṗvble(x"\1"y"\1") & x=
```

The string literal "\1" denotes the string " . Thus the value of x"\1"y"\1" is the value of x followed by " followed by the value of y followed by " . Remembering that spaces and division to lines are only for simplifying reading and not part of the real string, we see that the value of x"\1"y"\1" is Gödel's sentence. Furthermore, ~Pvble(x"\1"y"\1") claims that x"\1"y"\1" is not provable. To summarize, the other parts of Gödel's sentence make x"\1"y"\1" be Gödel's sentence, and ~Pvble(x"\1"y"\1") says that it is not provable. Altogether, Gödel's sentence claims that Gödel's sentence is not provable.

The formula Pvble() specifies a proof system for strings. Gödel's sentence is not a single sentence, instead, each proof system for strings has its own Pvble() and thus its own Gödel's sentence. Gödel's sentence of a proof system for strings claims that Gödel's sentence of that system is not provable in that system.

The Turing machine that halts immediately independently of the input represents a proof system for strings that proves every sentence. This proof system is useless, because for any sentence that it proves, it also proves its negation. So it proves many false sentences. However, it serves as an example of a proof system that proves its own Gödel's sentence.

Consider now any proof system for strings that proves its own Gödel's sentence. Because the sentence claims that the system does not prove it, the system has proven a false sentence. Consider then any proof system for strings that does not prove its own Gödel's sentence. Its Gödel's sentence thus expresses a true claim, and is thus a true sentence that the system does not prove.

We have proven the following.

Theorem 1 *Each recursively enumerable proof system for the first-order language on finite character strings with string literals, concatenation, and =, either proves a false sentence or fails to prove a true sentence.*

That is, there is no recursively enumerable proof system for strings that proves precisely the true sentences and nothing else. No recursively enumerable proof system for strings can precisely capture the true claims on strings that can be expressed in our language. This is the incompleteness theorem for strings.

7 Incompleteness of Natural Number Arithmetic

In this section we show that natural number arithmetic can simulate strings and their concatenation, and conclude that also natural number arithmetic is incomplete.

Our language on natural number arithmetic uses the same characters as our language on strings in Section 3. A *number literal* is either `0` or any non-empty finite sequence of digits that does not start with `0`. A *variable* is any string that starts with a lower case letter and then consists of zero or more digits. The value of a variable is a natural number. A *term* is a variable, a number literal, or any of the following, where t and u are terms: `(t)`, `t+u`, or `t*u`. The parentheses are used in the familiar way, `+` denotes addition, and `*` denotes multiplication. Furthermore, `*` has higher precedence than `+`, that is, `t+u*v` denotes the same as `t+(u*v)`. Atomic propositions and formulae are defined like in Section 3.

We now introduce a one-to-one correspondence between strings and natural numbers. Let $p = 53$, and let the 53 characters in the character set be given numbers from 1 to 53. If c'_i is a character, then let its number be denoted with c_i. The string $c'_1c'_2\cdots c'_n$ has the number

$$num(c'_1c'_2\cdots c'_n) \quad = \quad c_1p^{n-1}+c_2p^{n-2}+\ldots+c_{n-2}p^2+c_{n-1}p+c_n \quad .$$

So the empty string has the number 0, and the number of any string consisting of precisely one character is the number of that character. Let ι_n denote the string of length n whose every character has number 1. We have $num(\iota_n) = p^{n-1}+\ldots+p+1$, and ι_0 is the empty string.

We have to show that this mapping is indeed one-to-one. To do that, for each string $c'_1c'_2\cdots c'_n$ we introduce a *successor* and prove that the number of the successor is always one bigger than the number of the string itself. If $c_i = p$ for every $1 \leq i \leq n$, then the successor is defined as ι_{n+1}. We have

$$\begin{array}{r} num(\iota_{n+1}) \\ -\,num(c'_1c'_2\cdots c'_n) \end{array} \; = \; \begin{array}{r} p^n + \;\; p^{n-1} + \;\ldots\; + \quad p+1 \\ -\,pp^{n-1} - pp^{n-2} - \;\ldots\; - \;p\cdot 1 \end{array} \; = \; 1\,.$$

In the opposite case, at least one of $c_1, \ldots, c_n$ is not p. Let j be the last such index, that is, $1 \leq j \leq n$, $c_j \neq p$, and $c_i = p$ when $j < i \leq n$. The successor is defined as the string $d'_1d'_2\cdots d'_n$, where $d_i = c_i$ when $1 \leq i < j$, $d_j = c_j+1$, and $d_i = 1$ when $j < i \leq n$. We have

$$\begin{array}{l} num(d'_1d'_2\cdots d'_n) - num(c'_1c'_2\cdots c'_n) = \\ \quad\;\; c_1p^{n-1} + \ldots + c_{j-1}p^{n-j+1} + (c_j+1)p^{n-j} \qquad\qquad\quad + \;\; p^{n-j-1} + \ldots + \quad p+1 \\ \quad -\,c_1p^{n-1} - \ldots - c_{j-1}p^{n-j+1} - \qquad\;\; c_jp^{n-j} - pp^{n-j-1} - pp^{n-j-2} - \ldots - p\cdot 1 \\ =\; 1\,. \end{array}$$

We see that the empty string, its successor, the successor of that string, and so on are in one-to-one correspondence with the natural numbers 0, 1, 2, and so on. It remains to be proven that this sequence of strings covers all strings. It does not contain any string twice, because the corresponding natural numbers are all distinct. So it contains infinitely many distinct strings. For any n, there is only a finite number of strings of length n. So the sequence cannot get stuck at any length n. The only case where the successor is of different length than the string itself is when the successor is ι_{n+1}. So the sequence covers at least ι_0, ι_1, ι_2, and so on. Between ι_n and ι_{n+1}, including ι_n but not ι_{n+1}, the sequence goes through

$num(\iota_{n+1}) - num(\iota_n) = p^n$ strings of length n. The number of strings of length n is p^n, so the sequence goes through all of them.

We have shown that our correspondence between strings and natural numbers is one-to-one.

Our next task is to represent concatenation of strings as a formula on their numbers. The definition of *num* yields immediately

$$num(c'_1 \cdots c'_n d'_1 \cdots d'_m) \ = \ p^m num(c'_1 \cdots c'_n) + num(d'_1 \cdots d'_m) \quad .$$

To present this in our language, we have to extract p^m from $num(d'_1 \cdots d'_m)$ only using the language. Let $y = num(d'_1 \cdots d'_m)$. We have $num(\iota_m) \leq y < num(\iota_{m+1})$, that is, $p^{m-1} + \ldots + 1 \leq y < p^m + \ldots + 1$. Multiplying this by $p-1$ we get $p^m - 1 \leq y(p-1) < p^{m+1} - 1$, to which adding $y+1$ yields $p^m + y \leq yp + 1 < p^{m+1} + y$. If $m' > m$, then $p^{m'} + y \leq yp + 1$ does not hold, and if $m' < m$, then $yp + 1 < p^{m'+1} + y$ does not hold. Therefore, $k = p^m$ if and only if k is a power of p and $k + y \leq yp + 1 < pk + y$.

A prime number is a natural number greater than 1 that cannot be represented as a product of two natural numbers greater than 1. If p is a prime number and $p^m = xy$, then, for some $0 \leq i \leq m$, $x = p^i$ and $y = p^{m-i}$. Therefore, and because 53 is a prime number, the property that k is a power of 53 can be formulated as follows.

```
Pow53(k)  :⇔  ( Ax:Ay: k=x*y -> x=1 | Ez: x=53*z )
```

That $x < y$ can be expressed as follows.

```
lt(x, y)  :⇔  ( Ei: y=x+i+1 )
```

Based on these considerations, if `x` and `y` are the numbers of two strings, then the number of the concatenation of the strings is obtained as follows.

```
Cat(x, y, z)  :⇔
     ( Ek: Pow53(k) & lt(y*53+1, 53*k+y) & ~lt(y*53+1, k+y) & z=k*x+y )
```

Atomic propositions in our language on strings are of the form $t_1 \cdots t_m$=$u_1 \cdots u_n$ or $t_1 \cdots t_m$≠$u_1 \cdots u_n$, where $t_1, \ldots, t_m, u_1, \ldots, u_n$ are variables or string literals. They can be replaced as shown below for `=` , where `t` and `u` are two variable names that are different from the t_i and u_j.

```
( Et:Eu: t=u
     & ( Eu: Cat(u, $t_m$, t) & Et: Cat(t, $t_{m-1}$, u) & Eu: Cat(u, $t_{m-2}$, t) & ... )
     & ( Et: Cat(t, $u_n$, u) & Eu: Cat(u, $u_{n-1}$, t) & Et: Cat(t, $u_{n-2}$, u) & ... )
)
```

There is a Turing machine T_1 that inputs a sentence in the language on strings, replaces each string literal by its number, and replaces each atomic proposition as shown above. (We could have made this easier for the Turing machine but harder for the reader by, in Section 3, not allowing more than one character in any string literal, not allowing more than one variable and/or string literal in a term, and instead declaring that `<x+y:z>` expresses that `xy=z`.) If there is a Turing machine T_2 that halts on the true sentences in the language on natural numbers and fails to halt on false sentences, then there is a Turing machine T that first runs T_1 and then runs T_2 on the result. By construction, T halts if and only if its input string is a true sentence in the language on strings. But we proved in Section 6 that such a Turing machine does not exist. Therefore, T_2 does not exist. We have proven the following.

Theorem 2 *Each recursively enumerable proof system for the first-order language on natural numbers with* 0, 1, $+$, $\cdot$, *and* $=$, *either proves a false sentence or fails to prove a true sentence.*

8 Versions of Gödel's First Incompleteness Theorem

We proved that any recursively enumerable theory of natural numbers with zero, one, addition, multiplication, equality, logical and, logical not, and the universal quantifier either proves a false sentence or fails to prove a true sentence. Although this theorem is widely called Gödel's first incompleteness theorem, it falls short of what Gödel presented in [2]. It assumes that the truth or falsehood of a sentence can be reasonably talked about, even if the theory does not prove either. (When we say that a theory proves a sentence false, we mean that the theory proves the negation of the sentence.) This assumption has been criticized. Perhaps for this reason, Gödel went beyond this version. To discuss this, we first make the following observation.

In the presence of truth and falsehood as we usually consider them, a sentence and its negation cannot both be true. Furthermore, for each sentence, either it or its negation is true. A theory is *consistent* if and only if in no case it proves both a sentence and its negation. A theory is *complete* if and only if in each case it proves the sentence or its negation. (The word "complete" is used in more than one meaning in mathematical logic. This is the meaning we use here.) Therefore, if a theory proves only true sentences and proves all of them, then it is consistent and complete.

The notions of consistency and completeness do not rely on a pre-defined notion of truth of a sentence. However, they do not together mean the same as "proves only true sentences and proves all of them", because it may be that the theory fails to prove a true sentence and instead proves its false negation. Indeed, there are consistent and complete theories whose language is the same as the language of natural number arithmetic. An example is obtained by letting 0 denote false, 1 denote true, $+$ denote $\vee$, and $\cdot$ denote $\wedge$, and by adopting the usual axioms and inference rules of $=$ and propositional logic together with two special rules: "$\forall x : P(x)$ is equivalent to $P(0) \cdot P(1)$" and "$\exists x : P(x)$ is equivalent to $P(0) + P(1)$". This theory could well be called a first-order theory of truth values. It proves sentences that are false from the point of view of natural number arithmetic, such as $1 + 1 = 1$ (which represents $\mathsf{true} \vee \mathsf{true} = \mathsf{true}$). It is a consistent and complete theory, but a wrong theory for natural number arithmetic although it has the same language.

Another way to look at this is that the replacement of the notions of truth and falsehood by completeness and consistency disconnect the language from natural numbers, leaving only two uninterpreted constant symbols 0 and 1, and two uninterpreted binary operator symbols $+$ and $\cdot$. The mere fact that the symbols look familiar does not give them any formal properties. Instead, to make them again have a link with natural number arithmetic, some axioms and inference rules are needed.

In conclusion, the right liberation of the incompleteness result from the notion of pre-defined truth is that no "sufficiently strong" theory of natural number arithmetic is consistent and complete. Here "sufficient strength" has two aspects. First, the notion of first-order theories has a standard set of logical axioms and inference rules. It is assumed. Second, enough properties of natural numbers are assumed in the form of axioms, to ensure that the theory indeed is a theory of natural numbers instead of, say, the theory of truth values sketched above. Not much is needed. A rather weak axiom system known as *Robinson arithmetic* suffices [10, 12]. It is otherwise the same as the well-known Peano arithmetic, but the induction axiom has been replaced by the axiom "each natural number is either 0 or the result of adding 1 to some natural number."

Gödel did not prove the theorem in the form stated above. Instead of consistency, he used the stronger notion called *ω-consistency*. A theory is not ω-consistent if and only if it is not consistent or, for some formula P with one free variable x, it proves both $\exists x : P(x)$ and each one of $\neg P(0)$, $\neg P(1)$, $\neg P(2)$, $\ldots$. That a theory proves each one of $\neg P(0)$, $\neg P(1)$, $\neg P(2)$, $\ldots$ does not necessarily imply that it proves their conjunction $\forall x : \neg P(x)$, because no proof can go through an infinite number of cases one by one (proofs must be finite), and a common pattern that would facilitate proving them simultaneously in a single proof does not necessarily exist. Even so, intuition says that if none of $P(0)$, $P(1)$, $P(2)$, and so on holds, then there is no x such that $P(x)$ holds, that is, $\neg\exists x : P(x)$ holds. So a healthy theory of natural number arithmetic must be not only consistent, but also ω-consistent.

Gödel's result was that such a theory cannot be complete. Let $\mathsf{Prf}(x,y)$ denote the claim that natural number x is the encoding of a proof of the sentence encoded by natural number y. This claim can be formulated in natural number arithmetic. Furthermore, if $\mathsf{Prf}(x,y)$ holds, then $\mathsf{Prf}(x,y)$ can be proven, and if $\neg\mathsf{Prf}(x,y)$ holds, then $\neg\mathsf{Prf}(x,y)$ can be proven. Together with the requirement of "effectivity", this implies that the proof system must be recursive in the sense of Section 2. That is, there is a mechanical test which, for any string, tells whether it is a valid proof, where also the answer "no" is given explicitly instead of just never answering anything. Although this assumption is strictly stronger than recursive enumerability, proof systems typically satisfy it.

The proof system must also facilitate the simple reasoning steps in the sequel.

Gödel's self-referential sentence is $\neg\exists x : \mathsf{Prf}(x,g)$, where g is its own encoding as a natural number. Assume first that the proof system proves $\neg\exists x : \mathsf{Prf}(x,g)$. Then there is a natural number p that is the encoding of some proof of $\neg\exists x : \mathsf{Prf}(x,g)$. By the strength assumption above, the proof system proves $\mathsf{Prf}(p,g)$. From it the proof system can conclude $\exists x : \mathsf{Prf}(x,g)$. So it proves both $\exists x : \mathsf{Prf}(x,g)$ and its negation, and is thus not consistent. The case remains where the proof system does not prove $\neg\exists x : \mathsf{Prf}(x,g)$. Then no natural number is the encoding of a proof of $\neg\exists x : \mathsf{Prf}(x,g)$. By the strength assumption above, the proof system proves $\neg\mathsf{Prf}(0,g)$, $\neg\mathsf{Prf}(1,g)$, and so on. If the proof system is ω-consistent, then it does not prove $\exists x : \mathsf{Prf}(x,g)$. So it leaves both $\exists x : \mathsf{Prf}(x,g)$ and $\neg\exists x : \mathsf{Prf}(x,g)$ without a proof, and is thus incomplete.

Later Rosser found a modification to the proof that allows to replace ω-consistency with consistency [11]. We call his self-referential sentence R. It is $\forall x : (\mathsf{Prf}(x,r) \rightarrow \exists y : y \leq x \wedge \mathsf{Prf}(y,\overline{r}))$, where r is the encoding of R and $\overline{r}$ is the encoding of ¬R. If p is the encoding of a proof of R, then the system proves R, $\mathsf{Prf}(p,r)$, and $\exists y : y \leq p \wedge \mathsf{Prf}(y,\overline{r})$. If such an y indeed exists, then the proof whose encoding is y yields ¬R, so the system proves a contradiction. Otherwise, the system proves $\neg\mathsf{Prf}(0,\overline{r}) \wedge \neg\mathsf{Prf}(1,\overline{r}) \wedge \cdots \wedge \neg\mathsf{Prf}(p,\overline{r})$, yielding $\neg\exists y : y \leq p \wedge \mathsf{Prf}(y,\overline{r})$, a contradiction again. ω-consistency is not needed, because $\neg\mathsf{Prf}(0,\overline{r}) \wedge \cdots \wedge \neg\mathsf{Prf}(p,\overline{r})$ is a finite expression and thus a sentence.

If p is the encoding of a proof of ¬R, then the system proves $\mathsf{Prf}(p,\overline{r})$ and $\exists x : (\mathsf{Prf}(x,r) \wedge \neg\exists y : y \leq x \wedge \mathsf{Prf}(y,\overline{r}))$, yielding $\exists x : \mathsf{Prf}(x,r) \wedge \neg(p \leq x)$. Like above, the system proves one or another contradiction, depending on whether any of 0, 1, ..., $p-1$ is the encoding of a proof of R.

In conclusion, if the system is consistent, then it proves neither R nor ¬R, and is thus incomplete.

The above proof of Rosser's theorem uses the symbol $\leq$ that is not part of the first-order language on arithmetic. It can be expressed as mentioned in Section 2. The crucial property is that if c is a natural number constant, then no other natural numbers than 0, 1, ..., c have the properties that $x \leq c$ and $\neg(c+1 \leq x)$. This can be proven from Peano arithmetic, but in the case of other axiom systems, specific axioms on $\leq$ may be needed.

In his original publication [2] Gödel also sketched a proof of a corollary that is now known as Gödel's second incompleteness theorem. It says that natural number arithmetic does not prove its own consistency, if it indeed is consistent. What is more, no recursive consistent theory that contains natural number arithmetic proves its own consistency. The significance of this result is the following. Some mathematical principles are easy to accept, while some others have raised doubts. The questionable principles would become more acceptable, if, with a proof that only uses easily acceptable principles, they were proven to not yield contradictions. Gödel's second incompleteness theorem rules out perhaps not all, but at least the most obvious approaches to such proofs.

9 Related Work and Conclusions

We have shown that any recursively enumerable first-order theory of finite character strings with concatenation and equality either proves a false sentence or fails to prove a true sentence. From this we derived a similar result about natural number arithmetic with addition, multiplication, and equality, obtaining the "either proves a false sentence or fails to prove a true sentence" version of Gödel's first incompleteness theorem.

A *halting tester* is a Turing machine that reads any Turing machine M together with its input I and tells whether M halts, if executed on I. In addition to inventing his machines, Alan Turing proved that there is no halting tester [13]. (The modern version of this proof is very simple.)

Our proof of Gödel's theorem is based on encoding each claim of the form "M halts on I" as a sentence in natural number arithmetic. It is not the first such proof. If each encoded sentence "M halts on I" or its negation were provable by a system that only proves true sentences, then a halting tester would be obtained by letting a Turing machine test all finite character strings until it finds a proof of halting or non-halting. Therefore, the system has an unprovable true sentence. Essentially the same reasoning can be expressed in another words by pointing out that the set of provable sentences is recursively enumerable, but the set of true sentences is not recursively enumerable, because the true sentences of the form "M does not halt on I" cannot be enumerated. So truth and provability do not match. This proof is given in, e.g., [4, p. 354] (leaving the (A) mentioned below as a doubly starred exercise!) and [7, pp. 288–291], underlies the proof in [8, p. 134], and is at least hinted at in [6, p. 64].

Two major difficult technicalities in Gödel's proof are (A) to show that reasoning or computation can be encoded as properties of natural numbers (so-called *Gödel numbering*), and (B) to give a formula access to its own number. The proof based on non-existence of halting testers makes (B) trivial. It makes it necessary to check or believe that, given M and I, a Turing machine can perform the construction in (A). Fortunately, it is rather obvious.

Because natural number arithmetic has no direct construct for expressing finite sequences of natural numbers, (A) is surprisingly difficult. To do (A), often the Chinese Remainder Theorem is used. That brings discussion so far from the main topic that some expositions simply skip the issue. Alternatively, one may add the exponentiation operator n^m to the theory, as was done in [8, p. 135]. In [7], Dexter Kozen made (A) relatively easy by using a slightly less straighforward representation for halting computations than we did in Section 5, and treating natural numbers essentially as finite sequences of p-ary digits, where p is a prime. Numbers that were known to be powers of p, but not known which power, were used to extract individual digits. Thanks to padding with blanks, the representation of each configuration during a computation used the same number of digits. As a consequence, there was a number c such that if y extracts a digit in a configuration, then yc extracts the corresponding digit in the next configuration.

Our proof made (A) and (B) easy by doing them in a formalism that is very amenable to them. The most advanced number-theoretic property needed in the whole proof is that if a number is a power of prime p, then all its factors other than 1 are divisible by p. The Chinese Remainder Theorem was not used and the exponentiation operator was not added to the language. Turing machines were referred to twice: as the basis of the definition of "recursively enumerable proof system", and as devices that can perform a simple syntactic transformation.

It seems obvious that the incompleteness of theories of finite character strings can also be proven with the approach in [4, 7, 8]. Then one may continue like in Section 7. In this combined approach, the formula $\mathtt{Q(x, y)}$ would not be needed (but the computability of (A) by a Turing machine would).

Neil D. Jones has proven the incompleteness of first-order theories of nested lists with concatenation [5, p. 202]. Also this proof is based on the non-existence of halting testers. The counterpart of (A) is trivial, because the formalism supports it directly. Nested lists are a strong formalism that can easily express natural numbers, so this result is not surprising.

Finite character strings with concatenation may at first sight seem a poor formalism: a data type with infinitely many distinct values could not be much simpler. On the other hand, all computation reduces to the manipulation of finite character strings both in theory (Turing machines) and in practice (files are finite sequences of bytes). So the incompleteness of first-order theories of finite character strings with concatenation seems too obvious to be a new result. To prove it, it suffices to cite Gödel and then show that strings can simulate arithmetic. Such simulations have been studied at least in [1, 9]. However, neither publication explicitly mentions the incompleteness of strings and, indeed, the author has failed to find any mention of it in the literature. What is more, in this paper the proof was simplified by simulating in the opposite direction, that is, by proving the incompleteness of strings directly and then deriving the incompleteness of arithmetic as a corollary. This idea seems to be new.

Even if it turns out that the approach of this paper is not novel, we hope that the paper helps the readers understand Gödel's famous result.

Acknowledgements. This version of the paper has benefited from the good comments given by the anonymous reviewers.

References

[1] J. Corcoran, W. Frank & M. Maloney (1974): *String Theory*. *J. Symb. Log.* 39(4), pp. 625–637, doi:`10.2307/2272846`.

[2] K. Gödel (1931): *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I*. *Monatsh. Math. Phys.* 38(1), pp. 173–198, doi:`10.1007/BF01700692`. In German.

[3] D.R. Hofstadter (1979): *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books.

[4] J.E. Hopcroft & J.D. Ullman (1979): *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley.

[5] N.D. Jones (1997): *Computability and Complexity – From a Programming Perspective*. Foundations of computing series, MIT Press.

[6] S.C. Kleene (1943): *Recursive Predicates and Quantifiers*. *Trans. Amer. Math. Soc.* 53(1), pp. 41–73, doi:`10.1090/S0002-9947-1943-0007371-8`.

[7] D. Kozen (1997): *Automata and Computability*. Undergraduate texts in computer science, Springer, doi:`10.1007/978-1-4612-1844-9`.

[8] C.H. Papadimitriou (1994): *Computational Complexity*. Addison-Wesley.

[9] W.V. Quine (1946): *Concatenation as a Basis for Arithmetic*. *J. Symb. Log.* 11(4), pp. 105–114, doi:`10.2307/2268308`. Available at `http://projecteuclid.org/euclid.jsl/1183395170`.

[10] R.M. Robinson (1950): *An Essentially Undecidable Axiom System*. In: *Proceedings of the International Congress of Mathematics 1950*, pp. 729–730.

[11] J.B. Rosser (1936): *Extensions of Some Theorems of Gödel and Church*. *J. Symb. Log.* 1(3), pp. 87–91, doi:`10.2307/2269028`. Available at `http://projecteuclid.org/euclid.jsl/1183142131`.

[12] A. Tarski, A. Mostowski & R.M. Robinson (1953): *Undecidable Theories*. North Holland.

[13] A. Turing (1936): *On Computable Numbers, With an Application to the Entscheidungsproblem*. *Proceedings of the London Mathematical Society* 42, pp. 230–265. Correction in 43 (1937), 544–546.