Hilbert's Tenth Problem

Abhibhav Garg
150010

November 11, 2018

Introduction

This report is a summary of the negative solution of Hilbert's Tenth Problem, by Julia Robinson, Yuri Matiyasevich, Martin Davis and Hilary Putnam. I relied heavily on the excellent book by Matiyasevich, Matiyasevich (1993) for both understanding the solution, and writing this summary.

Hilbert's Tenth Problem asks whether or not it is decidable by algorithm if an integer polynomial equation has positive valued roots. The roots of integer valued polynomials are subsets definable in the language of the integers equipped with only addition and multiplication, with constants 0 and 1, and without quantifiers. Further, every definable set of this language is either the roots of a set of polynomials, or its negation. Thus, from a model theoretic perspective, this result shows that the definable sets of even very simple models can be very complicated, atleast from a computational point of view.

This summary is divided into five parts. The first part involves basic defintions. The second part sketches a proof of the fact that exponentiation is diophantine. This in turn allows us to construct universal diophantine equations, which will be the subjects of parts three and four. The fifth part will use this to equation to prove the unsolvability of the problem at hand. The final part has some remarks about the entire proof.

1 Defining the problem

A multivariate(finite) polynomial equation with integer valued coefficients is called a diophantine equation. Hilbert's question was as follows: Given an arbitrary Diophantine equation, can you decide by algorithm whether or not the polynomial has integer valued roots. For example, the Diophantine equation $X^2 + 1$ has no integer valued roots, while the polynomial $X^2 - 1$ does. Hilbert was not concerned with whether we can find the solution itself, just whether we can check if one exists.

For various reasons, it will be more convenient to consider only non-negative integer solutions. All existential quantifiers will thus range only over the naturals, unless expicitely mentioned otherwise. We can do this without loss of generality, since the problem in the general case is equivalent to the non-negative integers case. We can go from the first to the second by replacing the unknowns x_i by $y_i - z_i$, and can go from the second to the first using Lagrange's four square theorem. Further note that it does not matter whether we are looking for the solutions of a single polynomial, or a set of polynomials, since the latter can be converted to the former by taking a sum of squares of the set of polynomials.

We now define the notion of Diophantine sets. Consider any Diophantine eqution, $D(a_1, \ldots, a_n, x_1, \ldots x_n)$. Here, we have divided the set of unknowns into two parts: the parameters(the a_i) and the unknowns (the x_i). We call a set $P \subset \mathbb{N}^n$ a Diophantine set of dimension n if there is a Diophantine equation $D^P(a_1, \ldots a_n, x_1, \ldots x_m$ such that

$$(p_1, \ldots, p_n) \in P \iff \exists y_1, \ldots, y_m \left[D^P(p_1, \ldots, p_n, y_1, \ldots y_m) = 0\right]$$

In other words, a set is Diophantine of dimension n if there is a Diophantine equation with n parameters such that the equation has solutions if and only if the parameters are in the set. For example, consider the set of all even natural numbers. This is a Diophantine set of dimension 1, defined by the Diophantine equation $D(a, x) := 2x = a$. It is easily checked that a natural number solution exists if and only if a is an even natural number. It is also easily checked that Diophantine sets of the same dimension are closed under union (multiplication of their defining polynomials), intersection (sum of squares of their defining polynomials), and under applying an existential quantifier to one of the coordinates. We will later see that Diophantine sets are NOT closed under negation, and hence are not closed under for all quantifiers.

Having defined Diophantine sets, we now expand our vocabulary a little. We will call a property P Diophantine, if the set of all all natural numbers that satisfy this property is Diophantine. We have already seen than the property $IsEven$, which, as the name suggests, is true for the even natural numbers and false otherwise, is Diophantine. Similarly, one can see that $IsOdd$, $SmallerThanK$ for some fixed finite K are also Diophantine. Notice that the conjunction and disjunction of Diophantine properties is Diophantine, since we have already seen that the intersection and union of Diophantine sets are Diophantine. Looking at Diophantine-ness in terms of properties in this way is more natural, and makes it a lot easier to show that complicated sets are Diophantine - just show that the elements in the complicated set satisfy the a monotone boolean combination of some simple properties that are themselves Diophantine. From here on, we will do this freely.

Going further, we will call a $n - ary$ relation R Diophantine, if the n tuples that R defines form a Diophantine set. These will naturally be of dimension n. For example, the $2 - ary$ equality relation is diophantine: consider the polynomial $D(a_1, a_2, x) := x + (a_1 - a_2)^2 = 0$. If $a_1 = a_2$, then there is a natural number solution: $x = 0$. If not, then there is no natural number solution. Some other easy to check examples of Diophantine relations include inequalities $(\leq, \geq, <, >)$, and divisibility (all pairs (a, b) such that $a|b$).

Finally, we define a function to be Diophantine if its graph is a Diophantine set. For example, the $2-ary$ function $rem(b, c)$ which computes the remainder on dividing b by c is diophantine: $a = rem(b, c) \iff a < c \wedge c|(b - a)$. Here, we used the method described above - we wrote the required relation as a monotone boolean combination (in this case, a simple conjunction) of two relations that we had already showed are Diophantine. Already, writing the polynomial for the above relation is tedious. Eventually (starting basically now) our Diophantine sets will become so complex that actually writing the polynomials that define then will become infeasible, and we will solely rely on the method instead, slowly building up more and more complicated Diophantine sets. A slightly more non trivial example of this in action is that the greatest common divisor function is Diophantine. To see this, note that

$$a = gcd(b, c) \iff b > 0 \wedge c > 0 \wedge a|b \wedge a|c \wedge \exists x, y \left[a = bx - cy\right].$$

This also implies that the least common multiple is Diophantine.

In the upcoming section, we will show that exponentiation is Diophantine. This will let us allow exponentiatial polynomials in our definition of Diophantine set, which will further easily allow us to show that many other interesting sets, such as the set of all primes are Diophantine.

2 Exponentiation is Diophantine

The fact that exponentiation is Diophantine was the last missing piece of the negative resolution to Hilbert's Tenth. Conditioned on exponentiation being Diophantine, the fact that algorithms cannot decide if a polynomial has roots or not was proved in 1959. Subsequently, the condition of requiring exponentiation to be Diophantine was weakened by Robinson. Finally, in 1970, Matiyasevich proved that the Fibonacci numbers are Diophantine, which was enough to get exponetiation is Diophantine, thus finishing the proof.

Formally, we need that the set of three-tuples (a, b, c) that satisfy $a = b^c$ is Diophantine. The complete proof of this is very long, although not tedious. We refer the reader to the above mentioned book for the same, and sketch some ideas here.

2.1 Second Order Recurrent Sequences

Exponentiation can naturally be seen as a first order recurrent sequence, where the 1^{st} term is 1, and the n^{th} term is just the $(n-1)^{st}$ term multiplied by b. We instead now study a second order recurrence relation, defined as follows:

$$\begin{aligned}\alpha_b(0) &= 0\\ \alpha_b(1) &= 1\\ \alpha_b(n+2) &= b\alpha_b(n+1) - \alpha_b(n)\end{aligned}$$

It is easy to check that for $b > 2$, the above sequence grows roughly exponentially fast. In particular, $(b-1)^n \leq \alpha_b(n+1) \leq b^n$. We want to show that the set (a, b, c) such that $a = \alpha_b(c)$ is Diophantine. From here, we will easily be able to prove that the general exponential is Diophantine. In this report, we only sketch the proof that a projection of the above, namely the set (a, b) such that $a = \alpha_b(n)$ for some n is Diophantine. All of the hardness of the proof of exponentiation being Diophantine lies in using the fact that the projection is Diophantine to show that the originial set is Diophantine.

To show that the projection is Diophantine, notice that for all n, the values $\alpha_b(n-1)$ and $\alpha_b(n)$ satisfy the equation $\alpha_b(n-1)^2 - b\alpha_b(n-1)\alpha_b(n) + \alpha_b(n)^2 = 1$. This can be checked simply by induction. We claim that the following stronger property holds: for any solutions of the equation $x^2 - bxy + y^2 = 1$, along with the constraint that $x > y$, the solutions are of the form $(\alpha_b(n+1), \alpha_b(n)$ for some n. Proving this is slightly trickier, and can be done by inducting on y. In particular, it holds for $y = 0$. Further, given a solution (x, y), we can construct a new solution (y, z) with $z = by - x < y$, and thus use induction to say that y, z look like $\alpha_b(m'+1)$ and $\alpha_b(m')$ for some m', whence x, y have the same form for $m = m' + 1$. Then, the following formula is Diophantine and describes the necessary set: $b \geq 2 \wedge \exists x \left[x^2 - abx + a^2 = 1\right]$. If there is a solution x, then a has to be of the form $\alpha_b(m)$, and if a is of that form, then the witness for x is $\alpha_b(m+1)$.

2.2 Exponentiation From Second Order Recurrent Sequences

For convinience, let $0^0 = 1$. Consider the integer b^c. It can be verified via calculus, that

$$b^c = \lim_{x \to \infty} \frac{\alpha_{bx+4}(c+1)}{\alpha_x(c+1)}.$$

Not only does the above hold for all b, c, but it can also be checked that the sequence converges to the limit from above. We have already shown that the second order recurrent sequences are Diophantine, but the above formulation also has a limit operator, which is not contained in our language of Diophantine equations. However, using the fact that the limit converges from above, we can get rid of the limit. By the definition of limit, after some finite x, the sequence above is within a distance of 1 from b^c. At this point, if we round down, we can get the value of b^c. In other words, all we have to do is find a bound on x such that above this value, we can replace the division in the above definiton by integer divison (div function defined earlier), to get a function that evaluates to b^c. Some case bashing shows that we can use $x = 16(c+1)\alpha_{b+4}(c+1)$. Thus, we have that

$$a = b^c \iff x > 16(c+1)\alpha_{b+4}(c+1) \wedge a = \alpha_{bx+4}(c+1)\text{div}\alpha_x(c+1).$$

This shows that exponentiation is Diophantine. As a consequence, instead of looking at just polynomials, we can now allow exponential polynomials in our definition of Diophantine sets, without changing the problem at hand. This will allow us to encode arbitrary tuples of numbers into sinlge numbers, and decode the same, via Diophantine functions. This is done in the next section.

3 Coding

This part is a slight but necessary detour from the main flow of the proof. Here, we discuss methods of representing tuples of natural numbers as single natural numbers. Of course it is easy to see that the number

of tuples of finite lengths of natural numbers is countable, and hence there has to be a bijection between the two sets. However, for our purposes, a bijection will not be enough - we will require a Diophantine bijection. We discuss a few different codes here, each with their own advantages and disadvantages.

3.1 Cantor Coding

Consider the following linear ordering of pairs of numbers:

$$(0,0),(0,1),(1,0),(0,2),(1,1),(2,0),(0,3),\ldots$$

This is the ordering one usually sees in a first proof of the fact that the rationals are countable. Indexing this sequence starting with 0, one can see that the index of the pair (a,b) is given by

$$Cantor(a,b) = \frac{(a+b)^2+3a+b}{2}.$$

This is clearly a Diophantine function. Further, the function that takes in an index of a pair and outputs the first element of the pair is also Diophantine:

$$a = ElemF(c) \iff \exists y \left[(a+y)^2+3a+y = 2c\right].$$

where $ElemF$ is the function in question. Similarly, the function that projects on the second element is also Diophantine, making the Cantor code a very good way of representing pairs of integers.

The Cantor function can be generalised to represent n tuples of natural numbers in the natural way, as follows:

$$\begin{aligned} Cantor_1(a_1) &= a_1 \\ Cantor_{n+1}(a_1,\ldots,a_{n+1}) &= Cantor_n(a_1,\ldots,a_{n-1},Cantor(a_n,a_{n+1})) \end{aligned}$$

Further, we can check that the functions $Elem_{m,n}$, which project the index of n length tuples to the m^{th} coordinate is Diophantine. However, it is hard to show that the projection function is Diophantine as a function of m and n. Since we will eventually want to encode tuples of arbitrary length, Cantor coding will not be enough.

3.2 Positional Coding

In order to fix the issue with Cantor coding, we use positional coding. Assume that we want to encode the tuple $(a_1,\ldots,a_n)$ for an arbitrary n. We first pick a b such that $b > a_i$ for all i. Then, we set $a = a_n b^{n-1} + \ldots a_1 b^0$. In other words, we construct the number which when written in base b has our tuple as the digits. We use the triple (a,b,c) as our encoding, where we set $c = n$. Note that we originally wanted to get a single number as the encoding, but getting a constant tuple is good enough, since we can now use Cantor coding on the triple.

The advantage of positional coding is many fold. It is easy to see that the projection is Diophantine - to get the i^{th} number, we can just do integer division of a by b^{i-1}, and then take the remainder with respect to b. This is Diophantine as a function of both the input and the coordinate, which fixes the issue we had with Cantor coding. However, we have some further advantages, that will be crucial. Over the rest of this section, we list other operations on positional codes that are Diophantine, that will be useful later. An obvious operation is the concatenation of two tuples when written with the same base - this just involves multiplying by an appropriate power of b and adding. Less obvious is that arbitrary concatenation is Diophantine, as is comparision (elementwise) of two tuples, and application of a Diophantine function elementwise to a given tuple. The proof of this is too long to fit on this report, and the reader is refered to the book for the result. We do however list some intermediate steps of the proof of the above, which are of independent interest.

3.3 Some more examples of Diophantine sets

Consider the number $(b+1)^n$. It is easily checked that this is the a corresponding to the code of the tuple of binomial coefficients $\binom{n}{i}$, as long as b is chosen to be large enough. In particular, setting $b = 2^n + 1$, we get that $((2^n + 2)^n, 2^n + 1, n + 1)$ is a positional code for the binomial coefficients, whence arbitrary binomial coefficients are Diophantine, since they are obtained by projecting the above.

Further, note that we have

$$m! = \lim_{n\to\infty} \frac{n^m}{\binom{n}{m}}.$$

The above can be checked by simply substituting the denominator by the definition. We can now apply the same trick as before, to replace the limit operator by integer divison to get that

$$m! = (m+1)^{m(m+2)} \text{ div } \binom{(m+1)^{m+2}}{m},$$

which gives us that factorial is Diophantine.

Using the fact that factorial is Diophantine, and noticing that a number a is prime if and only if it is relatively prime with respect to $(a-1)!$, we get that the primality is Diophantine.

4 Universal Diophantine Equations

Now that we have seen how to encode arbitrary tuples of numbers into simple naturals in a Diophantine way, we can construct a Universal Diophantine Equation. This is a Diophantine equation with the universal property that solving any Diophantine equation reduces to solving projections of the Universal Diophantine equation. These will allow us to see construct our first non-Diophantine sets. Note that since the number of Diophantine equations are countable, but there are $2^{\aleph_0}$ many subsets of the natural numbers, that non-Diophantine sets exist is trivial. Proving something is non-Diophantine on the other hand, is a lot trickier.

4.1 Formal Definitions

A universal Diophantine equation for one-dimensional Diophantine sets is a Diophantine equation with the following form

$$U_1(a, k_1, \ldots k_l, x_1, \ldots, x_m) = 0.$$

The variables here are divided into three parts, a single parameter variable a, l many code variables $k_1, \ldots k_l$, and m many unknowns. Additionally, we require the following property: Given any one-dimensional Diophantine set P, defined by the Diophantine equation D, there exist numbers $k_1^D, \ldots, k_l^D$ such that

$$p \in P \iff \exists y_i \left[U_1(p, k_1^D, \ldots, k_l^D, y_i) = 0\right].$$

This is precisely the statement from before - for every Diophantine set P, there is a projection of U_1, such that the projected equation behaves exactly like the set P. In some sense, the Universal Diophantine equation is able to simulate any other Diophantine equation, when given appropriate description of the equation. Henceforth, Universal Diophantine equations is abbreviated as UDE. It is not hard to see that without loss of generality we can assume that $l = 1$. To convert from a UDE with more number of code parameters to one with fewer, we just replace the code parameters with unknowns, and add a second equation that forces these unknows to be the projections of a particular input Cantor code of appropriate dimension. Then, the single code can just be the Cantor coding of the original code. For example, given the above U_1, applying this transformation will give us the new UDE

$$U_1^2(a, z_1, \ldots, z_l, x_i) + (k - 2^{2^l} Cantor_l(z_i))^2 = 0,$$

where the k is the new code variable, and the z_i are new unknowns. Thus we can assume U_1 has $l = 1$. (In the above, the factor of 2^{2^l} is needed to make sure the coefficients are integers.

It is easy now to describe UDE for arbitrary dimension, the exact same definition as above works. Further, slightly more non trivial is the fact that we can start with U_1 and bootstrap this to get UDE for arbitrary n. For this, use a method similar to the one used to reduce the number of code parameters. We will not require UDEs for higher dimensions, and thus the details are skipped.

4.2 Constructing U_1

The consruction of the U_1 has many details which this report skips. The main idea behind the construction is to encode all the information about the polynomial into some code parameters, and also encode potential solutions. This encoding is will be done in a special way, that ensures that given the encoding of a polynomial, and the encoding of a solution, we will be able to check whether the polynomial evaluated on the solution is zero or not, in a Diphantine manner.

4.2.1 Encoding the equation

We encode the equation as a tuple of length 6, of the form (b, c_L, c_R, d, e, f). First we encode information about the number of unknowns and the total degree. We fix e to be the number of unknowns of the equation. We fix d to be any number greater than the total degree of the equation. Finally, we fix f as

$$f = 2^{d^1} + \cdots + 2^{d^e}.$$

While f does not provide any new information, and is a Diophantine function of d, e, it will make the evalution of the polynomial simpler. We call the triple (d, e, f) the format of the equation.

The equation in general will have terms with both positive and negative integers as coefficients. Since our codes can only deal with tuples of naturals, we write the given equation D as $D = C_L - C_R$, where C_L, C_R are equations with only positive coefficients (the L and R stand for left and right). To code a positive coefficient monomial, we need to order all monomials in a unique way, and code the tuple. The code we will use is as as follows:

$$c = \sum_{i_0+\cdots+i_m \leq d} i_0! \ldots i_m! (d - \sum i_j)! c_{i_0,\ldots,i_m} b^{d^{m+1} - i_0 d^0 - \cdots - i_m d^m}$$

This is just the positional code corresponding to a certain monomial ordering, where the coefficients of the polynomial have been scaled by different factors. The scaling is also done for reasons related to decoding, and it is easy to see that this scaling is done such that the original coefficients can easily be recovered. The above is defined as the cipher of the polynomial with respect to the base b.

It is now evident that in the original code, c_L and c_R will be the ciphers of the two parts of the original equation, and b will be the base. This completes the coding of the equation.

4.2.2 Coding solutions

Coding a solution amounts to simply coding the tuple $(x_1, \ldots, x_m)$. However, for the purpose of decoding, we do this is a slightly modified way. In particular, the code will consist of a tuple of length 5, of the form (d, e, f, g, h). The values d, e, f will just be the format of the equation for which we are coding the solutions. The number g will be any number bigger than all of the x_i. The number h is defined as

$$h := x_1 g^{d^1} + \cdots + x_m g^{d^m}.$$

Basically, h is the positional code of the tuple of the unknonws seperated by exponential numbers of $0's$, with base g.

4.2.3 Evaluating the polynomials

Given the code for a polynomial, and the code for a solution, at this point, it is fairly believable that evaluating the polynomial at the point is Diophantine. Showing all the details is slightly tedious and the details are skipped here. We formalise this entire notion, by defining a $9 - ary$ relation called *Solution*. In particular, the $Solution(a, b, c_L, c_R, d, e, f, g)$ is true if and only if (b, c_L, c_R, d, e, f) is a valid code for a polynomial, (d, e, f, g, h) is a potential solution for said polynomial, and the polynomial evaluated with parameter set to a, and unknowns set to those obtained by decoding the candidate solution equals 0.

4.2.4 Bringing it all together

We finally construct U_1. The relation *Solution* is Diophantine, whence there is a polynomial U corresponding to it, with 8 parameters, and unknowns $y_9, \ldots, y_m$, in particular of the form

$$U(a, b, c_L, c_R, d, e, f, g, y_9, \ldots, y_m) = 0.$$

Define U_1 to be

$$U_1(a, k, y_1, \ldots, y_m) := U^2(a, y_1, \ldots, y_m) + (k - 2^{2^6} Cantor(y_1, \ldots, y_6))^2.$$

The claim is that this is universal. First, we fix a code for each Diophantine equation P. Given an equation, encode it into the 6 tuple (b, c_L, c_R, d, e, f) as discussed into the previous part. Then, define its code k^P to be $2^{2^6} Cantor_6(b, c_L, c_R, d, e, f)$.

To see that this is universal, consider what happens when we substitute this code in the definition of U_1. Because of the second summand in the definition, the variables $y_1, \ldots, y_6$ are forced to take values equal to the encoding of the equation. Now, if the Diophantine set has the value p in it, then the original equation P has a solution when p is substituted in. In this case, we can take that solution and encode it. Then, we can say that U_1 has a solution for the same value of p - the witnesses for y_7, y_8 are f, g, the encoding of the solution. The other variables are witnessed by the definition of Solution. Similarly, the converse also holds. This completes the proof of the existence of U_1.

4.3 A Diophantine Set With Non-Diophantine Complement

We will use diagonalisation to construct such a set. The idea is as follows. Imagine a matrix with countably infinite rows and columns, each indexed by the naturals. Let the entry at i, j be equal to 1, if the polynomial $U_1(i, j, y_k)$ has solutions, and 0 otherwise. Each column of the matrix corresponds to a fixed code. Therefore, when we look at the column as an indicator vector, then the set it corresponds to is Diophantine set with that particular code. Further, by the universal property, every Diophantine set occurs as some column on this matrix - the one corresponding to its code.

Now we construct a set that is different from every row. Define $\mathfrak{H}$ to be the set of naturals a for which $U_1(a, a, y_i)$ has a solution. This is basically just the diagonal of our matrix. This is clearly a Diophantine set, the defining polynomial is just the one which on parameter p, is equal to $U_1(p, p, y_j)$.

Now consider the complement of this set $\bar{\mathfrak{H}}$. The claim is that this is not Diophantine. The proof is simple. Assume that it is Diophantine. Then it must have code k. Now consider the equation $U_1(k, k, y_j)$. If it has a solution, then by definition, $k \in \mathfrak{H}$. However, U_1 is universal, and since k is the code of $\bar{\mathfrak{H}}, k \in \bar{\mathfrak{H}}$. Assume then that we have no solution. Then by definition of $\mathfrak{H}, k \notin \mathfrak{H}$. But then, since U_1 is universal, $k \notin \bar{\mathfrak{H}}$. Since both situations lead to a contradiction, our hypothesis that the set is Diophantine is wrong.

5 Turing Machines and Diophantine Sets

For brevity, we assume that the reader is familiar with Turing machines. We denote our alphabet by A, the states by q_i, the state transition function by Q, the head movement funtion by D, and the write function by α.

5.1 Recursively Enumerable Sets

A set of natural numbers P is called *recursively enumerable* if there is a Turing machine M, such that when the input is a number $p \in P$, then $M(p)$ halts with sucess in finite time, and when the input is some $q \notin P$, then the machine M either halts with failure, or does not halt. It is not hard to see that any Diophantine set P (with equation D) is recursively enumerable. Given a input p, we can construct a machine that does the following: it enumerates all the tuples (y_j) of length equal to the number of unknowns, and evalutes $D(p, y_j)$. If the evaluation results in 0, it halts with success. Care needs to be taken that the order in which the tuples are enumerated is such that all tuples are hit in finite time. If $p \in P$, then the Turing machine will clearly halt - to check each tuple it takes finite time, and since all the tuples are well enumerated, the witness in the definition of Diophantineness will be hit in finite time. That the machine does not halt if $p \notin P$ is also clear. A more hard to believe result is that all recursively enumerable sets are Diophantine. The next subsection sketches the result of this fact.

5.2 All recursively enumerable sets are Diophantine

In order to show this, we show that the simulation of Turing machines is Diophantine. In particular, fix a Turing machine M. We want a way of fully describing the state of M during some arbitrary execution of M. To do this, we need to encode three things - the first is the tape, and second the position of the head, and the third is the state. Note that after any finite steps in time, only an initial segment of the tape is non-empty, of length say l. We can thus use the positional coding to encode the tape. To encode the other two bits of information, we do the following: construct a tuple of length l, that is 0 everywhere, except at the index which is equal to the position of the head. Here, the value of the tuple is i, where i is the state of the machine. We will call p the cipher of the second tuple (p stands for position), and t the cipher of the second string. Note that we can fix the base in advance, since we know the set of states and alphabet. Further, we do not need to encode the length, since p is guaranteed to have exactly one non-zero element, and t is actually supposed to encode an infinitely long sequence, since the empty sets are 0 by definition. We have thus encoded fully the machine M.

We now construct a Diophantine equation $D(p, t, x_i)$ that has a solution if and only if the parameters p, t take values such that when M starts at that state, it eventually halts.

For this, we start by showing that given a pair p, t the function that computes the position and tape ciphers after one step of the execution (call these functions $NextP, NextT$ are Diophantine. That $NextT$ is Diophantine is clear. Each element in the tuple $NextT(p, t)$ depends only on the corresponding elements in the tuples p, t. In particular, every element where p is 0 is just copied down unchanged. Where p is non zero, we can construct a function using the D, Q, α of M. Since all of these are finite, this function will be Diophantine. Finally, since positional coding allows function application, we are done.

That $NextP$ is diophantine is slighly more tricky, since the value at some cell depends not only on the corresponding values in p, t, but also on the values one index higher, and one index lower. To overcome this, given p, t we can create tuples p, t, p_l, p_r, t_l, t_r where p_l is the same as p, but shifted one index to the left, p_r is shifted to the right, and so on, via a Diophantine functon (positional codes allow this). Every index in $NextP$ is a finite function of the corresponding index in these 6 sequences, and thus $NextP$ is Diophantine.

To finish the construction of D, we will need to iterate over all possible number of steps for which we can run the machine. For this we need to define functions $NextPK(k, p, t)$ and $NextTK(k, p, t)$ that are able to take a parameter k, and simulate the machine for k steps. That these are Diophantine has a lot of detail, that are skipped here. This completes the constuction of D.

5.3 Hilbert's Tenth Problem is undecidable by Turing Machine

We have defined recursively enumerable sets. We now define *decidable* sets. A set of natural numbers is decidable P, if there is a Turing machine M such that given $p \in P$, the machine $M(p)$ halts in finite time with success, and given $q \notin P$, the machine $M(q)$ halts in finite time with failure. The difference from

recursively enumerable sets is that the machine is required to halt with the correct answer in both cases now.

If a set is decidable, then it is clearly recursively enumerable, the machine M from the definition of decidability is the required machine in the definition of recursive enumerability. It is also clear that the complement of the set is recursively enumerable. Here, the required machine is one that simulates M, and outputs the negation. This crucially relies on the fact that M always halts (and only on the existence of UTMs - but we have seen the existence of a UDEs, the fact that UTMs exist should be no surprise). Conversely, if a set and its complement are recursively enumerable, then the set is decidable. To construct the required machine, we do the following. Take the machines M' and M'' from the definition of recursive enumerability of the set and its complement. Define M to be the machine that simulates M' and M'', simultaneously (say one step of M', then a step of M'' then a step of M' and so on). It is easy to check that M always halts with the right answer.

We now reformulate Hilbert's Tenth Problem as follows: Can a Turing machine *decide* if a polynomial has a solution or not. Consider the set $\mathfrak{H}$ defined in the previous section. This set is Diophantine, whence recursively enumerable. We showed that its complement is not Diophantine, whence it is not recursively enumerable. Therefore, the set $\mathfrak{H}$ is not decidable.

What we have shown is that a Turing machine cannot even decide if a polynomial from a fixed family of polynomials has a solution or not. Thus, clearly a Turing machine cannot decide if an arbitrary polynomial has a solution or not.

6 Remarks

In order to just prove the undecidability of Hilbert's Tenth, we could have followed a slighly different route. In the above, we constructed a Diophantine set with a non-Diophantine complement. We could have instead skipped the entire fourth section, proved the equivalence of recursively enumerable sets and Diophantine sets, and could have constructed a recursively enumerable set with a non-recursively enumerable complement. This would have proved the same thing. Constructing this set would have involved constructing a universal turing machine, and doing essentially the same diagonalisation argument - encode the turing machines as natural numbers, and the matrix would have i, j entry 1 if turing machine i halts on input j with success, and 0 otherwise. This approach would have been a lot easier, and would have been the prefered one, had this been a computer science course.

I would like to point out (I came across this in a blog post by Terry Tao Tao, in particular Remark 16) that the above method is fairly general. All we really needed was that polynomials can simulate Turing machines. Similarly, it is known that nonlinear PDEs can simulate Turing machines. Therefore, given an arbitrary nonlinear PDE, we cannot decide by algorithm whether or not a smooth solution exists, because if we could, we can build a PDE such that solutions exist if and only if (insert any uncomputable problem here). There might also then a notion of a universal PDE.

References

Yuri V. Matiyasevich. *Hilbert's Tenth Problem*. MIT Press, Cambridge, MA, USA, 1993. ISBN 0-262-13295-8.

Terrence Tao. The "no self-defeating object" argument. URL `https://terrytao.wordpress.com/2009/11/05/the-no-self-defeating-object-argument/`.