Hilbert Meets Isabelle: Formalisation of the DPRM Theorem in Isabelle

Benedikt Stock, Abhik Pal, Maria Antonia Oprea, Yufei Liu, Malte Sophian Hassler, Simon Dubischar, Prabhat Devkota, Yiping Deng, Marco David, Bogdan Ciurezu, Jonas Bayer and Deepak Aryal

May 22, 2018

Hilbert Meets Isabelle*

Formalisation of the DPRM Theorem in Isabelle/HOL

Deepak Aryal[1], Jonas Bayer[1], Bogdan Ciurezu[1], Marco David[1], Yiping Deng[1], Prabhat Devkota[1], Simon Dubischar[2], Malte Sophian Hassler[1], Yufei Liu[1], Maria Antonia Oprea[1], Abhik Pal[1], and Benedikt Stock[1]

[1] Jacobs University Bremen gGmbH. Campus Ring 1, 28759 Bremen, Germany.
[2] Kippenberg-Gymnasium. Schwachhauser Heerstraße 62-64, 28209 Bremen, Germany.
[3] St. Petersburg Department of Steklov Mathematical Institute of Russian Academy of Sciences. 27 Fontanka, St. Petersburg, Russia.

Abstract. Hilbert's tenth problem, posed in 1900 by David Hilbert, asks for a general algorithm to determine the solvability of any given Diophantine equation. In 1970, Yuri Matiyasevich proved the DPRM theorem which implies such an algorithm cannot exist. This paper will outline our attempt to formally state the DPRM theorem and verify Matiyasevich's proof using the proof assistant Isabelle/HOL.

Keywords: Hilbert's tenth problem · DPRM Theorem · Isabelle · Diophantine equations · recursively enumerable

1 Background

In October 2017, Yuri Matiyasevich visited Jacobs University in Bremen, Germany. During his short stay, he gave a few talks on Hilbert's tenth problem and his negative proof of the problem. He was interested in a formal verification of the proof. And as a result of his visit, we as a small group of undergraduate students developed into the Hilbert–10 research group at Jacobs University Bremen under the supervision of Yuri Matiyasevich and Prof. Dierk Schleicher.

2 Hilbert's Tenth Problem and the DPRM Theorem

David Hilbert formulated the problem as follows: "Given a Diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: To devise a process according to which it can be determined in a finite number of operations whether the equation is solvable in rational integers."

Here, "rational integers" refer to integers in the normal sense and "Diophantine" can be simply replaced by "polynomial" since the other conditions in the statement already make it so. Further, "process" comes close to our modern understanding

* The project is being jointly supervised by Yuri Matiyasevich[3], Dierk Schleicher[1], and Bernhard Reinke[1].

of an “algorithm.” And before anything else, we should have a formal definition of an algorithm. One of the conventional ways of doing this, is via a Turing machine. An algorithm, then, can be described as a state table the encodes the transitions and start and halting conditions for a Turing machine that executes it. Matiyasevich’s proof presented in [1] uses register machines, which are equivalent to Turing machines.

In 1948, Martin Davis conjectured that every recursively enumerable set is Diophantine. We first define a few things to understand this equivalence.

Definition 1 (Recursively enumerable set). *A* recursively enumerable set *is a set A such that there exists an algorithm that halts at exactly the members of A. In the case of register machines, it is equivalent to say that A is accepted by some register machine.*

A related notion is that of a *recursive* or *computable* set. We call a set *recursive* or *computable* if there exists an algorithm that terminates (in finite time) only on inputs form this set. One then calls a set S computable if and only if there is a (total) computable function $f(x)$ such that

$$f(x) = \begin{cases} 1, & x \in S \\ 0, & x \notin S \end{cases}.$$

The notion of a *computable function* is informal — it is just a function whose value can be obtained using an “effective procedure”. However, one can formalize this notion using other equivalent modes of computations like register machines or Turing machines.

Definition 2 (Diophantine equation). *A Diophantine equation is a polynomial equation with integer coefficients and unknowns.*

Note that integer coefficients and unknowns can be further reduced to natural numbers[4]. Consequently, this project is only concerned with natural numbers. If not explicitly stated otherwise, all variables, parameters and coefficients will be non-negative.

Definition 3 (Diophantine set). *A Diophantine set is a subset A of $\mathbb{N}^i$ for some i such that there exists j and a Diophantine equation $D(x, y) = 0$ with $x \in \mathbb{N}^i$, $y \in \mathbb{N}^j$ such that $\forall a \in A$, $\exists b \in \mathbb{N}^j$, $P(a, b) = 0$ only for $a \in A$.*

Furthermore, a Diophantine equation can be generalised by including exponentiation, i.e. the exponents are allowed to be unknowns. instance, $x^{y+z^x} = 0$ is a valid exponential Diophantine equation. Consequently, one can define exponential Diophantine sets using an exponential Diophantine equation. Note that subtraction is not allowed in the exponent in order to make sure the equation always has an integer value.

[4] As a result of Lagrange’s theorem to express any natural number as the sum of four squares, for details refer to Section 1.2.1 of [1].

If a set is Diophantine, then we can find a Diophantine equation to "encode" that set. The set of squares is a simple example: it can be encoded by the equation $x - y^2 = 0$. Since relations (or functions) are also sets, we can use Diophantine equations to encode certain relations. One of them is the exponential relation R_{exp}: $\forall a, b, c,\ (a, b, c) \in R_{\text{exp}} \iff a = b^c$. One can show that if R_{exp} is Diophantine, then exponential Diophantine sets are Diophantine [1].

Recursive, or computable, sets are a proper subset of recursively enumerable sets[5]. Since Davis's conjecture implies that there exists a Diophantine set that is not computable, there can be no general algorithm to determine whether a Diophantine equation solutions in the integers. In other words, Hilbert's tenth problem has a negative solution if Davis's conjecture is true.

In 1950, Julia Robinson proved is there is a Diophantine set if there exists a Diophantine set $R \subset \mathbb{N}^2$ for which there is a function showing *roughly exponential growth*, then R_{exp} is Diophantine.

In a 1961 paper, Martin Davis, Hilary Putnam and Julia Robinson showed that every recursively enumerable set is exponential Diophantine.[2] Hence, the only step missing from proving Davis's conjecture is to find a Diophantine function that satisfies the Robinson predicates.

In 1970, Yuri Matiyasevich found the missing puzzle piece and thus proved Davis's conjecture, which is now usually known as the Davis-Putnam-Robinson-Matiyasevich (DPRM) theorem.

Theorem 1 (DPRM). *All recursively enumerable sets are Diophantine.*

In [1], Matiyasevich uses a slightly different approach from the original proof that makes it more suitable for formalisation.

3 Formalisation of Matiyasevich's Proof of the DPRM Theorem

The proof can be split into three broad parts: number-theoretical prerequisites, showing that exponentiation is Diophantine, and finally showing that a register machine can be simulated using exponential equations. These three parts correspond respectively to chapters two, three, and four in [1] — the main paper we follow for our formalisation.

Matiyasevich first shows that that exponentiation is Diophantine i.e. there exists a suitable polynomial P with the property:

$$p = q^r \iff \exists x_1, x_2, \ldots, x_m \colon P(p, q, r, x_1, x_2, \ldots, x_m) = 0\ .$$

He next shows that the halting condition of a register machine can be written as an exponential equation. Together, these two statements give us the required result: All listable sets are Diophantine and, consequently, there does not exist a general algorithm to determine whether a Diophantine equation has solutions in the integers.

[5] This follows from the halting problem. The construction of such a set can be done from a set $P = \{P_i\}$ of all programs by $\{2^i 3^x \mid \text{program } P_i \text{ halts on input } x\}$.

3.1 Number-theoretical prerequisites

To begin, note that intersections and unions of Diophantine sets of same dimension, inequality, equality, divisibility, and modulo are all Diophantine. Further, also the conjunction and disjunction operators are Diophantine. That is any of the aforementioned operations can be expressed with a polynomial. Consider for example the union of the two Diophantine sets A and B. Then,

$$a \in A \cup B \iff \exists x_1, \cdots, x_m \colon P(a, x_1, \ldots, x_m) = 0 \ .$$

Also note that a set A is Diophantine if one can find a system of polynomial equations

$$P_i(a, b) = 0, \ i \in \{0, \ldots, n\}$$

that have solutions whenever $a \in A$.

After having stated that these operators are Diophantine, we will now continue with some prerequisites that will facilitate describing register machines later. It is often convenient to work with the positional notation of a number a in some base b with the digits a_k i.e.

$$a = \sum_{k=0}^{\infty} a_k b^k \qquad (1)$$

where only finitely many a_k are non-zero.

We implement this first as a data type that stores the base b as a natural number and the digits a_k as numbers in a list.

```
datatype pos = Pos nat "nat list"
```

We next show that this data type is consistent with the number it represents by first checking if the k-th digits of both the data type a natural number are the same and that a list of numbers in a given base represents the correct number

```
lemma pn_consistency:
  fixes digits :: "nat list"
  fixes base :: nat
  assumes "base > 1"
  assumes "digits ~= []"
  assumes "pn_digit_smaller_than_base (Pos base digits)"
  shows "(n = pnval (Pos base digits))
         = ((Pos base (hl_clean_zeroes digits))
         = (pnconvert n base))"
        (is "?P = ?Q")

lemma pn_digit_equivalence:
  fixes digits :: "nat list"
  fixes base k :: nat
  assumes "base > 1"
  assumes "pn_digit_smaller_than_base (Pos base digits)"
  shows "(d = digit (pnval (Pos base digits)) base k)
          = (d = (digits!k))"
        (is "?P = ?Q")
```

In the particular case when the base $b = 2$ we consider two important relations — orthogonality and masking. We say that two numbers a and a are orthogonal when

$$(a \perp b) \iff a_k b_k = 0 \forall k \tag{2}$$

Using a `orthogonal` function that encodes this behaviour, it can be shown that

```
lemma lm02_41_ortho_odd_binomial:
  fixes a b :: nat
  shows "(orthogonal a b) = (odd ((a + b) choose b))"
  (is "?P = ?Q")
```

Similarly, we say that c masks b whenever $b_k \leq c_k$ for all k. Using Kummer's theorem and the fact that Binomial coefficients are Diophantine, we can find a Diophantine representation for the masking relation. This requires us to prove

```
lemma lm02_43_masking:
  fixes b c :: nat
shows "(masks c b) = (odd (c choose b))" (is "?P = ?Q")
```

Finally, using these relations we can show that digit-by-digit multiplication ($a \cdot b = c$ with $c_k = a_k \cdot b_k$) can be written as a generalised exponential Diophantine equation by proving:

```
lemma lm02_47_digit_mult:
  fixes a b c :: nat
  shows "(c = (digit_binary_mult a b))
        = ((masks a c)
        & (masks b c)
        & (orthogonal (a - c) (b - c)))"
```

3.2 Exponentiation is Diophantine

The first major part of the proof relies on the fact that a second order recurrence (similar to Fibonacci numbers) exhibits exponential growth and that exponentiation can be made Diophantine [6]

We first consider the sequence defined by

$$\alpha_b(0) = 0, \alpha_b(1) = 1, \alpha_b(n+2) = b\alpha_b(n+1) - \alpha_b(n).$$

and implement in Isabelle as

[6] The fact that exponentiation is Diophantine effectively proves Julia Robinson's hypothesis.

```
fun alpha :: "nat => nat => int" where
  "alpha b 0 = 0" |
  "alpha b (Suc 0) = 1" |
  "alpha\_n: "alpha b (Suc (Suc n)) = (int b) *
                                     (alpha b (Suc n)) -
                                     (alpha b n)"
```

Note that this sequence has several useful properties: it shows linear growth for $b = 2$ and grows exponentially with $b > 2$; $x = \alpha_b(m)$ and $y = \alpha_b(m+1)$ give us solutions to the Pell equation $x^2 - bxy + y^2 = 1$; and that it satisfies:

$$\alpha_b(k)|\alpha_b(m) \iff k|m, \alpha_b(k)^2|\alpha_b(m) \iff k\alpha_b(k)|m. \tag{3}$$

Once all these components are in place one can show that the relation between numbers a, b and c given by the formula:

$$3 < b \wedge a = \alpha_b(c) \tag{4}$$

is Diophantine by combining all the proprieties of the sequence α_b into a system of 15 equations with variables $(a,\ b,\ c,\ s,\ r,\ u,\ v,\ t,\ w)$ (for the actual system of equations refer to [1]). The implication goes in both directions: the system has solutions if relation (4) is satisfied (sufficiency) and, simultaneously, if Eq.4 is satisfied then one can find numbers s, r, t, u, v, w satisfying all the equations in the system (necessity).

We have found a Diophantine representation for something that grows exponentially. From here there is a small step to a generalisation of the representation for any exponential relation $p = q^r$.

$$p = q^r \iff \exists m, b : p < m \wedge q\alpha_b(r) - \alpha_b(r-1) \equiv p \mod m \tag{5}$$

with $m = bq - q^2 - 1$, $b = \alpha_{q+4}(r+1) + q^2 + 2$. For the particular cases $q = 0$, $r = 0$ and $p = 1$; and $q = 0$, $0 < r$ and $p = 0$ can be treated separately and then added to the final system of equations using the Diophantine operator conjunction.

The formalisation for this part of the proof is fairly straightforward and only required stating and simply proving the lemmas listed in the paper. For instance the main result (3.23)

$$\forall b \geq 2\ \forall k > 0 : \alpha_b(m) \equiv 0 \pmod{\alpha_b(k)} \Longleftrightarrow m \equiv 0 \pmod{k}, \tag{6}$$

of section 3.4 in [1], can be implemented using the previously defined function `alpha`:

```
theorem divisibility_alpha:
  fixes b k m :: nat
  assumes "b > 2" and "k > 0"
  shows "alpha b m mod alpha b k = 0 --> m mod k = 0"
   (is "?P --> ?Q")
```

The beginning of the implemented proof has the following form:

```
proof
  assume Q: "?Q"
    define n where "n = m mod k"
    from Q n_def have n0: "n=0" by simp
    from n0 have Abn: "alpha b n = 0" by simp
    from Abn divisibility_lemma1 assms(1) assms(2) n_def
    mult_eq_0_iff show "?P" by simp
  next assume P: "?P"
  [...]
qed
```

We make use of the predefined template: First, we assume `Q` and show that it implies `P` and afterwards assume that `Q` holds to prove `P`. This is a very basic and straight forward proof. In order to present a more sophisticated proof we introduce our own data type. It corresponds to 2×2 matrices which are used frequently in chapter 3.

```
datatype mat2 = mat (mat_11 : int) (mat_12 : int)
                    (mat_21 : int) (mat_22 : int)
```

The functions `mat_11, mat_12, mat_21, mat_22` give us access to the individual entries of our 2×2 matrices. Using these functions we can formulate a lemma which has a more interesting proof than the previous one. It has no direct correspondence in [1] but is a consequence of equation (3.38):

```
lemma congruence_Abm:
  fixes b m n :: nat
  assumes "b>2"
  defines "v == alpha b (m+1) - alpha b (m-1)"
  shows "mat_21 (mat_pow n (mat_pow 2 (A b m))) mod v
    = 0 mod v &
  mat_22 (mat_pow n (mat_pow 2 (A b m))) mod v
    = ((-1) ^ n) mod v"
  (is "?P n & ?Q n")
```

The statement is composed of two sub-statements (abbreviated with `?P n` and `?Q n`) which are connected with a logical and. The two statements are put in one lemma because the statements depend on each other: In the following proof by induction we need both `?P n` and`?Q n` to prove `?Q (Suc(n))`. Furthermore, for one step (`Q4`) there has to be made a case distinction on `m` because the case `m = 0` has to be treated separately.

```
proof(induct n)
case 0
  from mat2.exhaust have S1:
    "mat_pow 0 (mat_pow 2 (A b m)) =  mat 1 0 0 1" by simp
  then show ?case by simp
next
```

```
  case (Suc n)
  (* introducing abbreviations for matrices *)
  (* after that proof of P *)
  from [...] have F1: "?P (Suc(n))" by metis
  (* now proof of ?Q n, we use hypothesis on ?P n *)
  from Suc.hyps have Q1: "mat_22 (mat_pow n Z) mod v
    = (-1)^n mod v" by simp
  [...]
  (* now we need a a case distinction on m*)
  consider (eq0) "m = 0" | (g0) "m>0" by blast
  then have Q4: "h mod v = (-1) mod v"
  proof cases
    case eq0
    from eq0 have S1: "A b m = mat 1 0 0 1" by simp
    from eq0 v_def have S2: "v = 1" by simp
    from S1 S2 show ?thesis by simp
    next case g0
    (* this case is more involved *)
    [...]
    from S5 S8 S9 show ?thesis by simp
  qed
  [...]
  from [...] have F2: "?Q (Suc(n))" by simp
  from F1 F2 show ?case by blast
qed
```

In the inductive step, `?Q (Suc(n))` and `?P (Suc(n))` are proved independently and put together in a final step to finish the proof.

3.3 Simulation of register machines using equations

The second major part of the proof requires a mathematical description of register machines; in particular, it describes a method that can be used to simulate a register machine as a set of exponential equations.

The paper describes a register machine with an arbitrarily large number of registers $\mathrm{R}1, \mathrm{R}2, \ldots, \mathrm{R}n$. The machine executes a program with instructions labelled $\mathrm{S}1, \ldots, \mathrm{S}m$ where each $\mathrm{S}k$ can be of the type

$$\begin{aligned} \text{I } \mathrm{S}k &: \mathrm{R}l{+}{+}; \mathrm{S}i \\ \text{II } \mathrm{S}k &: \mathrm{R}l{-}{-}; \mathrm{S}i; \mathrm{S}j \\ \text{III } \mathrm{S}k &: \mathrm{HALT} \end{aligned}$$

Each of these instructions respectively increase the value of $\mathrm{R}l$ and move to $\mathrm{S}i$; decrease the value of $\mathrm{R}l$ when $\mathrm{R}l > 0$ and go to $\mathrm{S}i$, else move to $\mathrm{S}j$; and HALT.

The proof in [1] describes a "protocol" to handle the data of the register machine. This protocol can be viewed as three rectangular tables merged into one. The tables respectively handle the state, register, and zero-indicator data.

Each table has q columns where q is the number of instructions the machine executes before halting. The state table has m rows, one for each state, and the register and zero-indicator tables have n rows. Hence, for the t-th iteration of the machine $s_{k,t}$ represents the k-th state and $r_{l,t}$ the l-th register. The zero-indicator tables contains the "zero indicator" values $z_{l,t}$ such that $z_{l,t} = 0$ whenever $r_{l,t} = 0$ and one otherwise (See an example of a protocol chart in section 4.3 of [1]).

The register machine checks if a given value a belongs to a listable set and halts if and only if this value is accepted. That for a to be accepted by the machine there exists a sequence of state transitions such that

$$s_{m,q} = 1 \land s_{1,q} = ... = s_{m-1,q} = 0. \tag{7}$$

In [3] Xu, Zhang und Urban describe an implementation of a Turing machine in Isabelle. We use adapt their ideas to implement register machines for our formalisation.

We first describe the state of the machine as its own data type where `Add`, `Sub` and `Halt` respectively refer to the states transitions R$l++$, R$l--$, and Halt

```
datatype instruction =
  Add register state |
  Sub register state state |
  Halt
```

Both the `Add` and `Sub` instructions refer to arbitrary register Rl, which is meant to be the "index" of registers here. We call a list of all register values the *Tape* of the register machine, inspired by the metaphor used while describing Turing machines. The `state` are implemented in a similar fashion.

We call a column of the protocol the *configuration* of the register machine i.e, the "snapshot" of all current register values and encode it in the `configuration` data type. Since every unique state can be characterised by a specific instruction and the state of register values at the given "time" (iteration) of the register machine, we create a type synonym for the configuration as the 2-tuple:

```
type_synonym configuration = "(instruction * tape)"
```

The execution of the register machine is described by transitioning from one configuration to the next. This requires us to first "fetch" the next instruction from the program, given the register value:

```
fun fetch :: "program => instruction => nat => instruction" where
  "fetch p (Add r next) val = p!next" |
  "fetch p (Sub r next nextalt) val = p!(if val = 0
                                         then nextalt
                                         else next)" |
  "fetch p Halt val = Halt"
```

and then updating the tape based on the new instruction:

```
fun update :: "tape => instruction => tape" where
  "update t (Add r _) = list_update t r (t!r + 1)" |
  "update t (Sub r _ _) = list_update t r
                              (if t!r = 0
                               then 0
                               else t!r - 1)" |
  "update t Halt = t"
```

Where the function `list_update` changes the values of the registers. Combining these, we finally describe one “step” of the register machine:

```
fun step :: "configuration => program => configuration"
  where
    "(step (s, t) p) = (let nexts = fetch p s
                                      (p!s)
                                      (read t (p!s));
                            nextt = update t (p!s)
                        in (nexts, nextt))"
```

A natural next goal here is to use these functions to describe the actual “protocol” for the register machine and prove subsequent lemmas regarding it.

In particular, once we have a description of the rows of the protocol — values that give us the “history” of a particular state or register value over the execution time of the machine — we can describe them using the positional notation described earlier in Eq. 1. For each of the states, register values, and zero indicators, we can define

$$s_k = \sum_{t=0}^{\infty} s_{k,t} b^t \quad r_l = \sum_{t=0}^{\infty} r_{l,t} b^t \quad z_l = \sum_{t=0}^{\infty} z_{l,t} b^t \tag{8}$$

Here the base b is chosen such that it’s larger than any value that appears in the protocol. For some appropriate c and using masking relations (which were already proven to be Diophantine) we can describe Eq.7 as a system of exponential equations:

$$b = 2^{c+1} \quad s_m = b^q \tag{9}$$

The final value of s_m clearly depends on the register values r_l and the zero indicators z_l. The converse is also true here, that is, given numbers $s_1, \ldots, s_m$, $z_1, \ldots, z_n$, $r_1, \ldots, r_n$ and p, q, b, c that satisfy the above conditions, then the program will stop after q steps given a certain input a.

This equivalence shows that the register machine will terminate after q steps if and only if there exists a system of Diophantine equations that characterises the listable set accepted by the machine! And since exponentiation is Diophantine, every listable (recursively enumerable) set is indeed Diophantine!

4 Conclusion and Future Plans

When we started in October 2017, a computer verification of the entire DPRM theorem had seemed like too ambitious of a project. However, in the six months that have followed, we've made significant progress towards a full formalisation. Most of the theorems in Chapter 3 ("Exponentiation is Diophantine" in [1]) have been stated and proved, we've finished stating all the lemmas from Chapter 2 ("Number theoretical prerequisites" in [1]), and finished the required formalisation of a register machine in Isabelle (Chapter 4; "Simulation of register machines using equations"). We expect to finish the formalisation by the end of May and then refine things as required later.

At the time of writing, we weren't able to find any formally verified proofs of Hilbert's problems. When finished, our attempt would be the first such verification. One of the major outcomes of our work will be a complete computer verification of the DPRM-theorem. In the process, we also plan to produce a proof of Kummer's theorem, and produce a formalisation of a register machine as described in [1].

In the larger mathematical context, we see our attempt as bringing the methods of twenty-first century — formal verification — to one of the major results from the last century. We also see our work as an ode to Hilbert's problems and to more than two decades worth of work that Martin Davis, Hilary Putnam, Julia Robinson, and Yuri Matiyasevich put in to solve Hilbert's tenth problem.

References

1. Matiyasevich, Y.: On Hilbert's Tenth Problem. Lecture notes from the Pacific Institute for the Mathematical Sciences (2000), http://www.mathtube.org/lecture/notes/hilberts-tenth-problem.
2. Robinson, J.: Collected Works of Julia Robinson. American Mathematical Society (1996)
3. Xu, J., Zhang, X., Urban, C.: Mechanising Turing Machines and Computability Theory in Isabelle/HOL. Springer Berlin Heidelberg (2013)