Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS

Martin Blicha

Model building in polynomial representation

Department of Algebra

Supervisor of the bachelor thesis:	doc. RNDr. David Stanovský, Ph.D.
Study programme:	Computer Science
Specialization:	General Computer Science

Prague 2015

I would like to thank my supervisor doc. RNDr. David Stanovský, Ph.D. for showing me not only the beauty of algebra, but also deep mathematical connections where I would not expect them.

Název práce: Konstrukce modelů v polynomiální reprezentaci

Autor: Martin Blicha

Katedra: Katedra algebry

Vedoucí bakalářské práce: doc. RNDr. David Stanovský, Ph.D., Katedra algebry

Abstrakt: Hledání konečných modelů je problém definovaný takto: k dané teorii prvořádové logiky najít její konečný model. V této práci předkládáme nový způsob řešení tohoto problému, který je založen na překladu axiomů na soustavu polynomiálních rovnic. Existence modelu dané velikosti je pak ekvivalentní existenci řešení této soustavy. Ukážeme korektnost tohoto přístupu, implementujeme algoritmus založen na tomto přístupu a porovnáme jeho výkon s aktuálně nejlepšími systémy na hledání modelů.

Klíčová slova: hledání konečných modelů, polynomiální rovnice o více neznámých, konečné teleso, polynomiální reprezentace

Title: Model building in polynomial representation

Author: Martin Blicha

Department: Department of Algebra

Supervisor: doc. RNDr. David Stanovský, Ph.D., Department of Algebra

Abstract: Finite model finding is a problem defined as follows: given a theory in first-order logic, find a finite model of this theory. We present a new approach to finite model finding based on a translation of axioms to a system of multivariate polynomial equation. Existence of a model of given size is then equivalent to an existence of a solution of the system. We prove the correctness of this approach, implement it and compare its performance with current state-of-the-art model finders.

Keywords: finite model finding, multivariate polynomial equations, finite field, polynomial representation

Contents

Introduction

Finite model finding in first-order logic is an established area of research in the field of automated theorem proving. Its main goal is searching for counter-examples when detecting non-provability of given formula. This is most important in the area of formal verification where a counter-example points to a bug in the system being verified.

Many different approaches have been tried for finite model finding and they compete annually at The CADE ATP System Competition[1] (CASC, see (Sutcliffe and Suttner 2006))

The aim of this thesis is to implement and test a new approach proposed by Stanovský, the supervisor of this thesis. He found out that when looking for a model of size n, the clauses can be translated to a system of multivariate polynomial equations such that the set of clauses has a model of size n if and only if the system of equations has a solution. Moreover, the model can be constructed from the solution. In this thesis we prove the correctness of this approach, implement a model finder based on this approach and compare it to the state-of-the-art model finders.

This thesis is organized as follows. In Chapter 1 we describe the problem of finite model finding and give a survey of current approaches. In Chapter 2 we formally define the translation of clauses to multivariate polynomial equations, prove the correctness of this approach and introduce the basic algorithm for finite model finding using this approach. In Chapter 3 we discuss implementation details and tweaks to the basic algorithm and also present the results from testing the model finder.

Preliminaries and notation

In the theoretical part of the thesis, we work a lot with finite fields, polynomial rings over finite fields, and structures for the first-order logic, especially (Tarski's definition of) evaluation of formulas in structure under a variable assignment. We assume the reader is familiar with these notions and is able to work with them at basic level.

Our notation should be standard. Letters x, y stands for variables while a denote elements of a structure and (logical) terms are denoted by t, usually with index. q is reserved to denote a prime power number. We write $x_1, \dots, x_n$ to denote n-tuple, but if the actual count is not relevant or known from context, we use $\mathbf{x}$ or $\mathbf{a}$. $\mathbf{D}$ denotes a structure of first-order logic and D denotes its domain. Similarly when working with finite fields we use $\mathbf{F}$ to denote the field and F to denote just its domain. $\mathbf{F}[x_1, \dots, x_n]$ stands for a polynomial ring over the field $\mathbf{F}$. Sometimes we use $GF(q)$ when referring to a field of size q. We have reserved P, usually with a subscript, to denote polynomials and thus predicate symbols are denoted by R. Similarly G, H are used for function symbols, as F is used for fields.

[1]`http://www.cs.miami.edu/~tptp/CASC/`

1. Finite Model Finding Problem

In this chapter we give a brief description of the Finite Model Finding (FMF) problem and discuss a couple of techniques and algorithms designed to solve this problem.

1.1 Description of FMF

An instance of FMF is a set of formulas in the language of first-order logic, a first-order theory. The goal is to find a finite model of the theory (a finite structure in the language of the theory such that all axioms of the theory are satisfied in the structure) or to report that no such model exists. In this thesis we only consider the instances where the axioms are clauses (or equivalently, universal closures of clauses). This can be done without the loss of generality, as every theory can be translated to an *equiconsistent* theory consisting of clauses only. This process is called *Skolemization* and consists of converting the formula to its prenex form and then removing existential quantifiers by replacing occurrences of their variable by fresh functions (or constants).

FMF appears in the literature also under a few other names, for example, Finite Model Computation, Finite Model Building, Finite Model Construction. It seems no consensus has been reached; nevertheless, it is obvious they refer to the same type of problems. We chose the neutral word "finding" as the others seems to emphasize the constructive part of the problem and sometimes it is enough, and also easier, just to decide if the given problem has a model instead of explicitly constructing it.

1.2 Applications

As mentioned earlier, FMF is an established area of research in the field of Automated Theorem Proving. Its importance is mainly in constructing counter-examples to a given hypothesis. This has been successfully applied in Mathematics and Logic, Automated Deduction, and Program Verification (Caferra et al. 2004, p. 309).

1.3 Current methods for FMF

Several methods for finding finite models have been developed since its beginnings. According to Gebser et al. 2011; Claessen and Sörensson 2003, these methods can be divided into two major categories: translational and constraint solving approaches.

In the translational approach, an instance of FMF is encoded into a problem of a different type, which is then solved by means available for that problem. The solution for the original problem is computed from the solution for the encoding problem. The first and very popular method was to translate FMF to a satisfiability problem in a propositional logic and use the powerful machinery of SAT solvers to obtain the solution. This method originated in McCune 1994 who implemented it in his tool MACE. Another successful tool based on this method, PARADOX, was introduced in Claessen and Sörensson 2003 and significantly improved the techniques used in this approach. Beside propositional logic, a translations to function-free clause logic (Baumgartner et al. 2007) and to (incremental) Answer Set Programming (Gebser et al. 2011) were introduced and implemented in tools FM-Darwin and iClingo, respectively.

In the constraint solving approach, the search is performed directly on the problem without translating it. It is typically a basic backtracking search backed up by powerful constraint propagation methods using a *symmetry reduction* principle to avoid searching for isomorphic models. The main representative of this approach is J. Zhang and H. Zhang's (1995) tool SEM.

In this thesis we examine an algebraic approach to FMF, which falls into the translational category. We show that it is possible to convert a set of clauses to a system of *multivariate polynomial equations* over a finite field of size q such that there is a finite model of size q for the clauses if and only if the system of equations has a solution. Moreover, it is possible to construct the model from the solution.

2. An algebraic approach to FMF

In this chapter we present a way to represent a finite structure for a language of first-order logic in a finite field and show a basic algorithm that converts an instance of FMF to a system of polynomial equations over a finite field. Behind the core idea is the fact that for a finite field $\mathbf{F}$, any function $f\colon F^n \longrightarrow F$ can be represented by a polynomial P_f of n variables with coefficients from the field $\mathbf{F}$ and a bounded degree. By this representation we mean that $\forall (x_1,\ldots,x_n) \in F^n$ $f(x_1,\ldots,x_n) = P_f(x_1,\ldots,x_n)$ holds.

Recall that a finite structure $\mathbf{D}$ for a language L is defined as a finite set D (the *domain* of the structure) together with realizations of every predicate and function symbol of L, where realization of a function symbol of arity k is a function from D^k to D and realization of a predicate symbol of arity l is a subset of D^l or equivalently a function from D^l to $\{\top, \bot\}$. Moreover, if we have at least two elements in the structure, we can denote two distinct elements to represent the truth values and treat even realization of predicate symbol as if its range was D. It follows that given a bijection between a structure $\mathbf{D}$ and a finite field $\mathbf{F}$, realizations of the logical symbols can be represented as multivariate polynomials over $\mathbf{F}$. Note that the previous sentence indicates a little setback. For some finite cardinalities, there is no finite field of that cardinality. Therefore, we will now deal only with cardinalities of prime powers and show how to deal with the others later.

2.1 Translating clauses to polynomial equations

The idea of translating a set of clauses to polynomial equations was inspired by the fact that every function in finite field can be represented by a polynomial over this field with bounded degrees of variables. This follows from the known interpolation theorems for finite fields.

Theorem 2.1 (Interpolation theorem for finite fields)**.** *Let $\mathbf{F}$ be a finite field of size q and h be a function from F to F. Then there exists exactly one polynomial $P_h \in \mathbf{F}[x]$ with $deg(P_h) < |F|$ such that $\forall a \in F$ $P_h(a) = h(a)$.*

Proof. This is a direct consequence of general interpolation theorem. Lagrange polynomial $\sum_{i=1}^{q} h(a_i) \cdot \prod_{j \neq i} \frac{x-a_j}{a_i-a_j}$ is the wanted unique polynomial representing h. □

Theorem 2.2 (Interpolation theorem for finite fields in higher dimensions)**.** *Let $\mathbf{F}$ be a finite field of size q and h is a function from F^n to F. Then there exists a polynomial $P_h \in \mathbf{F}[x_1,\ldots,x_n]$ such that $\forall \mathbf{a} \in F^n$ $P_h(\mathbf{a}) = h(\mathbf{a})$.*

Proof. We will prove this by induction on n with the base case $n = 1$ already covered by Theorem 2.1.

For the inductive step we need the following generalization of Theorem 2.1: Suppose h is a function from F to R where R is an integral domain extending F. Then there again exists a unique polynomial $P_h \in R[x]$ representing h. This holds because the corresponding Lagrange polynomial is a member of $R[x]$.

Now suppose we are looking for a representing polynomial for $h\colon F^{n+1} \longrightarrow F$. Consider the following q functions obtained by fixing the last argument of h:

$$h_{a_1}\colon F^n \longrightarrow F \text{ where } h_{a_1}(\mathbf{x}) = h(\mathbf{x}, a_1)$$

$$\vdots$$

$$h_{a_q}\colon F^n \longrightarrow F \text{ where } h_{a_q}(\mathbf{x}) = h(\mathbf{x}, a_q)$$

Using the inductive hypothesis, we obtain polynomials $P_{h_{a_1}}, \dots, P_{h_{a_q}}$ representing functions $h_{a_1}, \dots, h_{a_q}$. Since these polynomials are members of $\mathbf{F}[x_1, \dots, x_n]$, we can define a function $h' : F \longrightarrow R$ (where $R = \mathbf{F}[x_1, \dots, x_n]$) as follows: $\forall i\ h'(a_i) = P_{h_{a_i}}$. Then we use the generalized version of Theorem 2.1 to obtain $P_{h'} \in R[x_{n+1}] = \mathbf{F}[x_1, \dots, x_n, x_{n+1}]$ such that $\forall i\ P_{h'}(a_i) = P_{h_{a_i}}$. It is easy to see that $P_{h'}$ is the representing polynomial of h. □

Note than we can put boundaries on the degree of representing polynomials. In finite field F of size q, it holds that $\forall x \in F\ x^q = x$. This means that we can consider only representing polynomials where exponents of variables are less than q. Such polynomials have at most q^k monomials, where k is the arity of represented logical symbol, as each monomial is a distinct combination of k variables with degrees in range $[0; q-1]$. For example, a binary function h over $GF(2)$ has a representing polynomial of the form

$$P_h(x_1, x_2) = c_0 + c_1x_1 + c_2x_2 + c_3x_0x_1 \tag{2.1}$$

where $\forall i\ c_i \in GF(2)$. Actually, we know that that *every* binary function over $GF(2)$ has a representing polynomial of the form as in 2.1, varying only in the coefficients c_i. This means that if we abstract from the concrete values of the coefficients and treat c_i as unknowns - *coefficient variables*, than 2.1 can be regarded as a representing polynomial for the binary *function symbol h*, not only for a particular function. When looking for a particular realization of the symbol, we just have to provide the right values of the coefficients.

We already mentioned predicates can also be viewed as functions, therefore the case of predicate symbols is the same as the case of function symbols. However, the notion of representing polynomial is not restricted just to stand-alone function and predicate symbols. It can be extended to arbitrary terms and even to clauses.

Definition 2.1. Let $\mathbf{F}$ be a finite field of size q and let φ be a clause. If ρ is a translation mapping each function and predicate symbol in φ to its representing polynomial over $\mathbf{F}$ then we define a translation τ to recursively translate φ to a polynomial in a following way:

- For a disjunction, it recursively translates its disjuncts and then multiplies the results: $\tau(\xi \vee \psi) = \tau(\xi) \cdot \tau(\psi)$.
- For a negation, it recursively translates the inner formula and subtract 1 from the $(q-1)$-th power of the result: $\tau(\neg\psi) = \tau(\psi)^{q-1} - 1$.
- For an equality, it recursively translates terms on each side and subtract one from the other: $\tau(t_l = t_r) = \tau(t_l) - \tau(t_r)$.

- For a predicate symbol or a function symbol, it recursively translates the arguments and substitutes the results in the representing polynomial of the symbol: $\tau(S(t_1, \dots, t_k)) = \rho(S)(\tau(t_1), \dots, \tau(t_k)$.
- For a variable, it returns this variable as a polynomial: $\tau(x) = x$.

We illustrate the translation on a simple example.

Example 2.1. Consider a clause in a language with one constant a and one binary predicate symbol R:

$$R(x, a) \vee \neg R(x, x)$$

Suppose we look for a model with only two elements, so we will work in $GF(2)$. First, define how the symbols R and a will be translated:

- $\rho(a) = c_1$,
- $\rho(R(x, y)) = c_2 + c_3x + c_4y + c_5xy$.

Now proceed to translate the atomic formulas, the negation, and finally the disjunction. Properties of $GF(2)$ can be used to simplify the polynomials as much as possible:

- $\tau(R(x, x) = c_2 + c_3x + c_4x + c_5x^2 = c_2 + c_3x + c_4x + c_5x$,
- $\tau(R(x, a) = c_2 + c_3x + c_4c_1 + c_5c_1x$,
- $\tau(\neg R(x, x)) = \tau(R(x, x)) - 1 = c_2 + c_3x + c_4x + c_5x + 1$,
- $\tau(R(x, a) \vee \neg R(x, x)) = \tau(R(x, a)) \cdot \tau(\neg R(x, x)) = c_2^2 + c_2c_3x + c_2c_4x + c_2c_5x + c_2 + c_2c_3x + c_3^2x^2 + c_3c_4x^2 + c_3c_5x^2 + c_3x + c_1c_2c_4 + c_1c_3c_4x + c_1c_4^2x + c_1c_4c_5x + c_1c_4 + c_1c_2c_5x + c_1c_3c_5x^2 + c_1c_4c_5x^2 + c_1c_5^2x^2 + c_1c_5x =$
  $= c_2c_4x + c_2c_5x + c_3c_4x + c_3c_5x + c_1c_2c_4 + c_1c_3c_4x + c_1c_4x + c_1c_4 + c_1c_2c_5x + c_1c_3c_5x$.

Notice that in the resulting polynomial there are two distinct kinds of variables. There are variables that were already present in the clause itself and were preserved in the translation process (as seen in the last point of the definition of τ). We will refer to these as *element variables.* And then there are unknown coefficients, introduced by translating function and predicate symbols. These will be referred to as *coefficient variables.*

Definition 2.2. Let φ be a clause, $\mathbf{F}$ a finite field of size q and $\tau(\varphi)$ the translation of φ with respect to $\mathbf{F}$. Then $tr_{in}(\varphi) = \{\tau(\varphi)[e] \mid e \colon VAR(\varphi) \longrightarrow F\}$ is the *instantiated translation* of φ.

Instantiated translation is simply the set of polynomials obtained from $\tau(\varphi)$ by instantiating all element variables in $\tau(\varphi)$ with every possible combinations of values. That is, every member of $tr_{in}(\varphi)$ is obtained by taking an assignment of element variables e and substituting element variables in $\tau(\varphi)$ with values assigned by e.

Moreover, let $eq(\varphi)$ denote the system of equations obtained from $tr_{in}(\varphi)$ simply by setting each polynomial from $tr_{in}(\varphi)$ to be equal to zero. Note that unknowns of the systems are precisely the coefficient variables of $\tau(\varphi)$.

Example 2.2. To illustrate the instantiation, consider the situation from Example 2.1. The clause was translated to a polynomial

$$c_2c_4x + c_2c_5x + c_3c_4x + c_3c_5x + c_1c_2c_4 + c_1c_3c_4x + c_1c_4x + c_1c_4 + c_1c_2c_5x + c_1c_3c_5x.$$

There is only a single element variable x, and two possible values (since we were working in $GF(2)$). Substituting these values yields two polynomials with only the coefficient variables left.

- $x := 0 \Longrightarrow c_1c_4 + c_1c_2c_4$
- $x := 1 \Longrightarrow c_2c_4 + c_2c_5 + c_3c_4 + c_3c_5 + c_1c_2c_4 + c_1c_3c_4 + c_1c_2c_5 + c_1c_3c_5$

It is possible to translate not only a single clause, but also a set of clauses. The translation of function and predicate symbols just need to be consistent, it has to use the same coefficient variables when translating the same symbol in various clauses. The instantiated translation of a set of clauses can then be used to verify if it has a model of given size (note that the translation of function and predicate symbols always depends on the field).

Theorem 2.3. *Let T be a set of clauses and $\mathbf{F}$ be a finite field of size q. Then T has a model of size q if and only if $eq(T)$ has a solution over $\mathbf{F}$.*

The following section is basically the proof of this theorem, but we believe the topic to be interesting enough to deserve a separate section.

2.2 Representing finite structures of first-order logic in finite fields

In this section we show that a finite field can be used to represent any finite structure of a prime power size. In the finite field representation we define logical notions of *value of term and formula* and *satisfiability*. Then we show that these are well-defined, that is it holds that

$$\mathbf{D} \vDash \varphi \text{ iff } \mathfrak{F} \vDash \varphi$$

where $\mathfrak{F}$ is the finite field representation of structure $\mathbf{D}$ an φ is a clause. At the end, we use the notion of finite field representation to prove Theorem 2.3.

Consider a finite first-order structure $\mathbf{D}$ such that $|D| = q$, where q is a prime power. Take a finite field $GF(q)$ and denote it $\mathbf{F}$. Then there is a bijection b between D and F. Fix one bijection $b : D \longrightarrow F$ for the rest of the section. Using b, it is possible to pull the functions and relations of $\mathbf{D}$ to F. For an n-ary function symbol H and its realization $H^{\mathbf{D}} : D^n \longrightarrow D$ the corresponding function $H^{\mathbf{F}} : F^n \longrightarrow F$ is defined as $H^{\mathbf{F}}(a_1, \ldots, a_n) = b(H^{\mathbf{D}}(b^{-1}(a_1), \ldots, b^{-1}(a_n)))$. Similarly, for an n-ary predicate symbol R and its realization $R^{\mathbf{D}}$ in $\mathbf{D}$, the corresponding relation $R^{\mathbf{F}}$ is defined such that $R^{\mathbf{F}}(a_1, \ldots, a_n)$ if and only if $R^{\mathbf{D}}(b^{-1}(a_1), \ldots, b^{-1}(a_n))$.

In Theorem 2.2 we showed that all functions over a finite field have their representing polynomials. Since we can treat relations as functions (by denoting some elements as $\top$ and some as $\bot$), we can replace the functions and relations over F by their representing polynomials. By doing this we obtain a finite field representation $\mathfrak{F}$ of the structure $\mathbf{D}$ consisting of a finite field $\mathbf{F}$ and a set of representing polynomials.

We show how to evaluate terms and clauses in such representation and then we prove that satisfiability behaves the same way in the finite field representation as in the original structure.

Definition 2.3 (Term evaluation)**.** The value of a term t in $\mathfrak{F}$ for a variable assignment $e\colon VAR \longrightarrow F$ is inductively defined as follows:

- *variable*: $x^{\mathfrak{F}}[e] = a$, where $x \in VAR$ and $e(x) = a$,
- *complex term*: $G(t_1, ..., t_n)^{\mathfrak{F}}[e] = P_G(t_1^{\mathfrak{F}}[e], ..., t_n^{\mathfrak{F}}[e])$ where P_G is the polynomial representing realization of G.

Lemma 2.1. *$t^{\mathfrak{F}}[e'] = b(t^{\mathbf{D}}[e])$ for $e' = b \circ e$.*

Proof. Let $e\colon VAR \longrightarrow D$ and $e' = b \circ e$ be variable assignments for $\mathbf{D}$ and $\mathfrak{F}$, respectively. Proceed by induction on the complexity of the term:

- *variable*:

$$\begin{aligned} x^{\mathfrak{F}}[e'] &= e'(x) && \text{(Definition 2.3)}\\ &= b(e(x)) && (e' = b \circ e)\\ &= b(x^{\mathbf{D}}[e]) && \text{(Tarski definition)} \end{aligned}$$

- *complex term*:

$$\begin{aligned} G(t_1, \ldots, t_n)^{\mathfrak{F}}[e'] &= P_G(t_1^{\mathfrak{F}}[e'], \ldots, t_n^{\mathfrak{F}}[e']) && \text{(Definition 2.3)}\\ &= P_G(b(t_1^{\mathbf{D}}[e]), \ldots, b(t_n^{\mathbf{D}}[e])) && \text{(induction hypothesis)}\\ &= G^{\mathbf{F}}(b(t_1^{\mathbf{D}}[e]), \ldots, b(t_n^{\mathbf{D}}[e])) && (*)\\ &= b(G^{\mathbf{D}}(t_1^{\mathbf{D}}[e], \ldots, t_n^{\mathbf{D}}[e])) && \text{(definition of } G^{\mathbf{F}})\\ &= b(G(t_1, \ldots, t_n)^{\mathbf{D}}) && \text{(Taski definition)} \end{aligned}$$

  $(*)$ - P_G is the representing polynomial of the function $G^{\mathbf{F}}$

□

We continue by defining the value of a formula in $\mathfrak{F}$. The value of a formula is, as in the case of terms, an element of F and we show how to interpret such evaluation as true or false to obtain agreement with the evaluation in the original structure.

Definition 2.4 (Formula evaluation)**.** The value of a formula φ in $\mathfrak{F}$ for the variable assignment $e\colon VAR \longrightarrow F$ is inductively defined as follows:

- *equality*: $(t_1 = t_2)^{\mathfrak{F}}[e] = t_1^{\mathfrak{F}}[e] - t_2^{\mathfrak{F}}[e]$
- *predicate*: $R(t_1, \ldots, t_n)^{\mathfrak{F}}[e] = P_R(t_1^F[e], \ldots, t_n^F[e])$ where P_R is the representing polynomial of R.

- *disjunction*: $(\psi_1 \vee \psi_2)^{\mathfrak{F}}[e] = \psi_1^{\mathfrak{F}}[e] \cdot \psi_2^{\mathfrak{F}}[e]$
- *negation*: $(\neg\psi)^{\mathfrak{F}}[e] = v^{|F|-1} - 1$, where $v = \psi^{\mathfrak{F}}[e]$.

Definition 2.5 (Satisfiability in finite field representation)**.** Let $\mathfrak{F}$ be a finite field representation, e an assignment of the variables, and φ a formula. We define the satisfiability of φ in $\mathfrak{F}$ under e as

$$\mathfrak{F} \models \varphi[e] \text{ iff } \varphi^{\mathfrak{F}}[e] = 0.$$

Lemma 2.2. *Let* $\mathbf{D}$ *be a first-order structure of prime power size* q *and let* $\mathfrak{F}$ *be its finite field representation. Then for every formula* φ *in the language of* $\mathbf{D}$ *and every variable assignment* $e\colon VAR \longrightarrow D$ *and* $e' = b \circ e$ *it holds that*

$$\mathbf{D} \models \varphi[e] \textit{ iff } \mathfrak{F} \models \varphi[e'].$$

Proof. Proceed by induction on the complexity of the formula:

- *equality*:

$$\begin{aligned} & \mathbf{D} \models (t_1 = t_2)[e] & \\ \iff & t_1^{\mathbf{D}}[e] = t_2^{\mathbf{D}}[e] & \text{(Tarski definition)} \\ \iff & b(t_1^{\mathbf{D}}[e]) = b(t_2^{\mathbf{D}}[e]) & (b \text{ is bijection}) \\ \iff & t_1^{\mathfrak{F}}[e'] = t_2^{\mathfrak{F}}[e'] & \text{(Lemma 2.1)} \\ \iff & t_1^{\mathfrak{F}}[e'] - t_2^{\mathfrak{F}}[e'] = 0 & \\ \iff & (t_1 = t_2)^{\mathfrak{F}}[e'] = 0 & \text{(Definition 2.4)} \\ \iff & \mathfrak{F} \models (t_1 = t_2)[e'] & \text{(Definition 2.5)} \end{aligned}$$

- *predicate*:

$$\begin{aligned} & \mathbf{D} \models R(t_1, \dots, t_n)[e] & \\ \iff & R^{\mathbf{D}}(t_1^{\mathbf{D}}[e], \dots, t_n^{\mathbf{D}}[e]) & \text{(Tarski definition)} \\ \iff & R^{\mathbf{F}}(b(t_1^{\mathbf{D}}[e]), \dots, b(t_n^{\mathbf{D}}[e])) & \text{(definition of } R^{\mathbf{F}}) \\ \iff & P_R(b(t_1^{\mathbf{D}}[e]), \dots, b(t_n^{\mathbf{D}}[e])) = 0 & (*) \\ \iff & P_R(t_1^{\mathfrak{F}}[e'], \dots, t_n^{\mathfrak{F}}[e']) = 0 & \text{(Lemma 2.1)} \\ \iff & R(t_1, \dots, t_n)^{\mathfrak{F}}[e'] = 0 & \text{(Definition 2.4)} \\ \iff & \mathfrak{F} \models R(t_1, \dots, t_n)[e'] & \text{(Definition 2.5)} \end{aligned}$$

  $(*)$ - P_R is the representing polynomial of R and 0 is the only element denoting $\top$

- *disjunction*:

$$\begin{aligned} & \mathbf{D} \models \psi_1 \vee \psi_2[e] & \\ \iff & \mathbf{D} \models \psi_1[e] \text{ or } \mathbf{D} \models \psi_2[e] & \text{(Tarski definition)} \\ \iff & \mathfrak{F} \models \psi_1[e'] \text{ or } \mathfrak{F} \models \psi_2[e'] & \text{(Induction hypothesis)} \\ \iff & \psi_1^{\mathfrak{F}}[e'] = 0 \text{ or } \psi_2^{\mathfrak{F}}[e'] = 0 & \text{(Definition 2.5)} \\ \iff & \psi_1^{\mathfrak{F}}[e'] \cdot \psi_2^{\mathfrak{F}}[e'] = 0 & \text{(Field property)} \\ \iff & (\psi_1 \vee \psi_2)^{\mathfrak{F}}[e'] = 0 & \text{(Definition 2.4)} \\ \iff & \mathfrak{F} \models \psi_1 \vee \psi_2[e'] & \text{(Definition 2.5)} \end{aligned}$$

- *negation*:

$$\begin{aligned}
&\mathbf{D} \models \neg\psi[e] \\
&\iff \mathbf{D} \not\models \psi[e] && \text{(Tarski definition)} \\
&\iff \mathfrak{F} \not\models \psi[e'] && \text{(Induction hypothesis)} \\
&\iff \psi^{\mathfrak{F}}[e'] \neq 0 && \text{(Definition 2.5)} \\
&\iff (\psi^{\mathfrak{F}}[e'])^{q-1} - 1 = 0 && \text{(Field property)} \\
&\iff (\neg\psi)^{\mathfrak{F}}[e'] = 0 && \text{(Definition 2.4)} \\
&\iff \mathfrak{F} \models \neg\psi[e'] && \text{(Definition 2.5)}
\end{aligned}$$

□

The correctness of the finite field representation is the key point in the proof of Theorem 2.3.

Proof of Theorem 2.3. Suppose a theory T has a model $\mathbf{D}$ of size q. Fix a clause $\varphi \in T$. Since $\mathbf{D}$ is a model of T it holds that $\forall e\ \mathbf{D} \models \varphi[e]$. Take a finite field representation $\mathfrak{F}$ of $\mathbf{D}$. By Lemma 2.2 it holds that $\forall e\ \mathfrak{F} \models \varphi[e]$. By Definition 2.5 this is equivalent to $\forall e\ \varphi^{\mathfrak{F}}[e] = 0$. Now consider the set of equations $tr_{in}(\varphi)$. There is an equation for every evaluation e and this equation is the result of translating φ by τ from Definition 2.1. Notice that τ mirrors the evaluation of terms and formulas in finite field representation (Definition 2.3 and 2.4). The only difference is that evaluation works with representing polynomials with *known* coefficients and τ works with representing polynomials with *unknown* coefficients. Since $\varphi^{\mathfrak{F}}[e] = 0$ for every evaluation e, this means that coefficients of representing polynomials in $\mathfrak{F}$ satisfy every equation in $tr_{in}(\varphi)$.

The proof of the other implication is practically the same. If a solution of the system of equations exists than using the solution as coefficients for the representing polynomials yields a finite field representation $\mathfrak{F}$ such that $\varphi^{\mathfrak{F}}[e] = 0$ for every evaluation e and every $\varphi \in T$. This follows from the fact that τ mirrors the evaluation of terms and formulas in such $\mathfrak{F}$ and for every clause φ every evaluation e is covered by one equation in $tr_{in}(\varphi)$. As a consequence $\mathfrak{F}$ represents a model of T. □

Example 2.3. At the end of Example 2.2 we obtained the instantiated translation $tr_{in}(\varphi)$ where $\varphi = R(x,a) \vee \neg R(x,x)$. By solving the corresponding system of equations $eq(\varphi)$ we obtain a two-element model for the original clause φ. For example, the trivial solution $\forall i\ c_i = 0$ translates to a finite model $\mathbf{M}$ with domain $M = \{0,1\}$ and with realizations of symbols as $a^{\mathbf{M}} = 0$ and $R^{\mathbf{M}} = M \times M$. Other solutions yield different models, for example a solution $c_3 = 1$ and $\forall i \neq 3\ c_i = 0$ yields realizations $a^{\mathbf{M}} = 0$ and $R^{\mathbf{M}} = \{[0,1]; [0,0]\}$.

2.3 Algebraic algorithm for FMF

In this section we use the ideas from the previous sections in an algebraic algorithm for finding finite models. First, we present a subalgorithm for finding a model of given prime power size, then we show how to deal with other sizes and finally we sum it all up and present a pseudocode for the algebraic algorithm for finding finite models.

2.3.1 Core subalgorithm

In Section 2.1 we showed, given a prime power q, how to translate a set of clauses T to a system of polynomial equations over finite field such that T has a model of size q if and only if the system of equations has a solution. Here we formulate this approach as an algorithm.

Algorithm 1 `FindModelBasic`(T, q)

```
Require: $q$ is prime power, $T$ is a set of clauses
 1: $eq(T) \leftarrow \emptyset$
 2: $\rho \leftarrow SymbolTranslation(T, q)$
 3: for $\varphi \in T$ do
 4:     $\tau(\varphi) \leftarrow Translate(\varphi, \rho)$
 5:     $tr_{in}(\varphi) \leftarrow \emptyset$
 6:     for $e$ in $\{e \mid e\colon VAR(\varphi) \longrightarrow GF(q)\}$ do
 7:         $tr_{in}(\varphi) \leftarrow tr_{in}(\varphi) \cup Instantiate(\tau(\varphi), e)$
 8:     end for
 9:     $eq(T) \leftarrow eq(T) \cup AsEquations(tr_{in}(\varphi))$
10: end for
11: return $Solve(eq(T))$
```

The input of this function consists of the set of clauses T and a prime power q determining the finite field $GF(q)$. At the beginning, a translation of predicate and function symbols from T to corresponding representing polynomials over $GF(q)$ is defined and used in the next phase to translate every clause from T as described in Definition 2.1. After that, all the translations are instantiated, i.e. element variables are substituted by elements of the field. Finally, every polynomial is transformed to equation by setting it to be equal to zero. The resulting system of equations is then solved and the algorithm returns the solution or reports that no solution exists.

If a solution of the system is found, the representing polynomials with coefficients from the solution together with the field $GF(q)$ represents a model of T of size q. If no solution exists than the clauses does not have a finite model of size q. Note that this works only for prime power sizes. It the next section we describe how to modify Algorithm 1 so it can be used also for sizes that are not prime power.

2.3.2 Modification for size that is not a prime power

The problem we are facing is that if we are looking for a model of size that is not a prime power, there is no finite field of that size. We can overcome this obstacle if we work in the closest bigger field and denote a subset of the domain of the field, with the wanted size, that will constitute the domain of the smaller model. Suppose we are looking for a model of size k which is not a prime power. Denote q to be the smallest prime power larger than k, $\mathbf{F}$ to be the finite field of size q and let S stand for a subset of F of size k, with $0 \in S$. Under some modifications, we can use the algorithm to found a structure in $\mathbf{F}$ (not necessary a model of the theory) which yields a model of size k after restricting the realizations of function and predicate symbols to S.

There are two steps where we need to modify the algorithm. First, we want the restriction to be a closed structure, meaning that for every function h it must be true that if $\forall i \; x_i \in S$, then $h(x_1, \ldots, x_n) \in S$ as well. Fortunately, this condition can be formulated as a system of equations. More precisely, it can be formulated as a set of clauses in the extended language using elements of S and these clauses can be translated to equations the same way as original clauses of the theory (translating the elements of S to themselves).

For each n-ary function symbol H a *restrictive* clause

$$\bigvee_{s \in S} H(x_1, \ldots, x_n) = s$$

is added to the original clauses. Restrictive clauses are translated the same way as original clauses and during the instantiation clauses, as we plan to restrict the realization of function symbols to S and thus do not care about the cases where one of the arguments is not from S, only elements from S are used for substituting.

Besides adding restrictive clauses, one more modification is necessary. If we would just add restrictive clauses and Algorithm 1 would successfully return a solution, then the resulting structure $\mathfrak{F}$ would be a model of size q and the restriction $\mathfrak{F}\restriction S$ would be a model of size k. However, it could happen that the bigger model does not exist but the smaller model does. In this case, Algorithm 1 would return failure and the smaller model would not be found. To ensure that this does not happen, we need to do one more modification.

Recall that during instantiation (lines 6 to 8) the translation of a clause is instantiated with all possible combinations of values from the field. We do this because the clause must hold for all elements of the model, in our case all elements of the field. In current situation, however, the model we are aiming for has domain S, not all F. So instead of substituting all possible combinations of values from F, we consider only those with all values from S. In the algorithm, on line 6, the variable e will cycle through elements of

$$\{e \mid e \colon VAR(\varphi) \longrightarrow S\}\,.$$

Now if a solution to the final system of equations is found, it yields a structure $\mathfrak{F}$ which is not necessarily a model of the theory, but the substructure $\mathfrak{F}\restriction S$ with domain S *is* a model of the theory. On the other hand, if a model of the smaller size exists, then there also exists its extension in a form of $\mathfrak{F}$, which satisfies the system of equations.

Incorporating these modifications results in a general algorithm for finding a model of a fixed size. For a size that is not a prime power, restrictive clauses are added to the theory (line 6) and assignments of variables have smaller range during instantiation (line 12).

Algorithm 2 `FindModelGeneral`(T, k)

```
Require: $k$ is integer $> 0$, $T$ is a set of clauses
 1: if $k$ is prime power then
 2:     return FindModelBasic$(T, k)$
 3: end if
 4: $q \leftarrow NextPrimePower(k)$
 5: $S \leftarrow SubsetOfSize(GF(q), k)$
 6: $T \leftarrow T \cup RestrictiveClauses(T, S)$
 7: $eq(T) \leftarrow \emptyset$
 8: $\rho \leftarrow SymbolTranslation(T, q)$
 9: for $\varphi \in T$ do
10:     $\tau(\varphi) \leftarrow Translate(\varphi, \rho)$
11:     $tr_{in}(\varphi) \leftarrow \emptyset$
12:     for $e$ in $\{e \mid e\colon VAR(\varphi) \longrightarrow S\}$ do
13:         $tr_{in}(\varphi) \leftarrow tr_{in}(\varphi) \cup Instantiate(\tau(\varphi), e)$
14:     end for
15:     $eq(T) \leftarrow eq(T) \cup AsEquations(tr_{in}(\varphi))$
16: end for
17: return $Solve(eq(T))$
```

2.3.3 Incremental algorithm

The final algorithm starts at size 1 and looks for models of increasing sizes until it finds one or until it runs out of resources.

Algorithm 3 `FindModel`(T)

```
Require: $T$ is a set of clauses
1: $k \leftarrow 1$
2: while TRUE do
3:     $sol \leftarrow$ FindModelGeneral$(T, k)$
4:     if $sol \neq \perp$ then
5:         return $ModelFromSolution(sol)$
6:     end if
7:     $k \leftarrow k + 1$
8: end while
```

2.3.4 Properties

Since algebraic approach is a translational approach to FMF, the best way to analyze it is by comparison to other translational approaches. We point out some similarities and differences between our and other approaches.

Firstly, although the search for the model is incremental, the algorithm is not able to share information between stages. This is a consequence of using different fields in different stages as the predicate and function symbols have different representing polynomials in different fields and all operations are evaluated differently, with respect to the current field.

Secondly, full instantiation is necessary in our approach. For every clause and every possible evaluation of its variables there is one equation. So there are k^n equations for a clause with n variables when looking for a model of size k. This indicates that the number of variables in a clause has great impact on the size of the resulting system of equations and beyond certain boundary the size can become intractable. Full instantiation was necessary both in SEM-style and MACE-style approach. PARADOX (Claessen and Sörensson 2003) was very successful in spite of this obstacle. However, recent approaches try to avoid this problem. This can be seen in FM-Darwin (Baumgartner et al. 2007) and iClingo (Gebser et al. 2011) where the full instantiation is not part of the translation.

Most translational approaches cannot handle nested terms, so a step called *flattening* is a part of the translation. It flattens nested terms at the cost of introducing new variables to clauses. This is particularly undesirable in PARADOX because of the full instantiation. In our approach the flattening is not necessary as nested terms are not an obstacle for the translation.

Since we translate the clauses to a system of equations, any solver can be used to tackle the equations. The solver can be completely independent of the translation process and thus one implementation can be easily replaced by another if it proves to be better. As a result, an advance in the field of solving multivariate polynomial equations directly translates to an improvement of our algorithm. This feature is common for all translational approaches (with respect to their target domain).

Some optimizations introduced by Claessen and Sörensson 2003 can be applied in our approach as well since they either operate on the clauses or introduce new ones. They are very well explained there, so here we discuss them only briefly. *Static symmetry reduction* is expressed as clauses using elements of the field, so it can be translated to equations. It introduces only few new equations and they are very useful since they restrict possible values of the variables. As a consequence, *sort inference* is also useful as it can strengthen the effect of symmetry reduction. The effect of *splitting* is double-edged. It splits a clause to multiple clauses having less variables, and usually also less literals, than the original clause. As a result, less equations are created during instantiation than with the original clause. However, it introduces new split predicates and this means that the resulting system of equations has more variables than the original one. The number of new variables depends on the arity of the split predicate and consequently on the number of variables shared by the split parts of the clause. Splitting a clause to two disjoint parts (with respect to shared variables) introduces only one new coefficient unknown (nullary predicate), but with more shared variables the number of new coefficient unknowns grows exponentially, and the savings in terms of equation count decreases as well.

3. Implementation and results

In this chapter we describe the implementation of the algebraic algorithm for finding finite models, test it on benchmarks from TPTP library (Sutcliffe 2009) and analyze the results.

3.1 Choosing software

The algorithm consists of two main parts, which are independent of each other. The first part is the translation of the input clauses to system of multivariate polynomial equations, the second is solving the resulting system.

Translation of clauses is relatively easy, it only requires manipulation with multivariate polynomials like multiplication, sum, substituting a value (or another polynomial) for a variable. These can be provided by a library or one can implement them by oneself in a favorite programming language.

On the other hand, solving a system of multivariate polynomial equations is much harder. This problem has been heavily studied due to its connection to cryptanalysis and several interesting algorithms have been proposed to solve it. Bard 2009 provides a nice survey of the problem and proposed solutions in two chapters. Chapter 11 focuses on some theoretical aspects of the problem and Chapter 12 describes algorithms used to solve this problem. The oldest and most popular approach is the computation of Gröbner basis. The notion of Gröbner basis was introduced by Bruno Buchberger in his PhD thesis in 1965 (english translation was published in 2006, see Buchberger 2006) and it proved to be very important tool in solving problems both in and outside of computer algebra. A nice online survey (Buchberger and Kauers 2010) was published in Scholarpedia[1]. Czech readers can find a nice explanation of Gröbner bases and ideas of the original Buchberger's algorithm for their computation in Stanovský and Barto 2011. There exist more than one variant of the algorithm for computing Gröbner basis of a set of multivariate polynomials, modern variants beating original Buchberger algorithm significantly. They are implemented in various software for computer algebra like Magma, Maple, Mathematica and SageMath(via SINGULAR). Besides Gröbner basis, other algorithms for solving multivariate polynomial equations were introduced, for example XL (for eXtended Linearization) algorithm (Courtois et al. 2000; Courtois 2001) and its variants. Recently a translation of polynomial equations over finite fields to SAT (first implemented in Bard et al. 2007) has become very popular. When choosing between this options, we ruled out the SAT approach, as the original problem (FMF) can be translated to SAT directly. From the others, we chose to use SageMath (Stein et al. 2015) with its component SINGULAR (Decker et al. 2015) for the computation of Gröbner basis. The main reason was that SageMath is an open-source project with one of the best implementations of Gröbner basis algorithm provided via SINGULAR. SageMath, with its Python based language, also provides user-friendly interface for operations with multivariate polynomials.

[1] `http://www.scholarpedia.org`

3.2 SageMath

SageMath, also referred to only as Sage, is a free open-source mathematics software system that supports research and teaching in algebra, geometry, number theory, cryptography, numerical computation, and related areas. Both the SageMath development model and the technology in SageMath itself are distinguished by an extremely strong emphasis on openness, community, cooperation, and collaboration. An interested reader can learn a lot about SageMath by reading its tutorial. [2]

Most of SageMath is implemented using Python and has several ways of usage. It can be used interactively from command line using its interactive shell, it also provides a very nice graphical interface called Notebook and finally it allows writing interpreted and compiled programs in Sage or writing stand-alone Python scripts that use Sage library.

For our purpose, it is important that SageMath provides a well-documented and user-friendly way to work with multivariate polynomials. SageMath distribution contains SINGULAR, a computer algebra system for polynomial computations, and it uses a shared library interface to SINGULAR called libSINGULAR to provide specialized and optimized implementations for multivariate polynomials over many coefficient rings, including finite fields. SINGULAR also provides one of the best implementations of an algorithm computing Gröbner bases.

3.3 Implementation

In this section we describe the details of our implementation of the algebraic algorithm for solving FMF.

As we said earlier, we decided to implement the algorithm using SageMath. Although its Notebook interface seemed appealing, we decided to implement it as a stand-alone Python script to enjoy the advantages of Python editor. The source code, together with user's guide, can be found in the attachment of this thesis. The algorithm requires SageMath to run, so it has to be installed[3] on your computer.

3.3.1 Data structures

SageMath provides several well-documented classes for work with multivariate polynomials, see the reference manual[4]. After creating the desired polynomial ring to work in, it is easy to create polynomials. They can be specified directly, or built step by step from ring generators (the variables) using sum, multiplication and other operations. SageMath also allows creating ideal from set of polynomials and calling the computation of Gröbner basis of this ideal in just two steps. This saves a lot of time as it is not necessary to code such classes from scratch. Unfortunately, it comes at a price. Working in the finite field $GF(q)$ allows using field equations (in the form of $x^q = x$) to reduce the degrees of variables in

[2]available at `http://doc.sagemath.org/html/en/tutorial/index.html`

[3]http://doc.sagemath.org/html/en/installation/index.html

[4]`http://doc.sagemath.org/html/en/reference/polynomial_rings/polynomial_rings_multivar.html`

polynomials if they exceed certain boundary. However, SageMath does not do this automatically, it strictly distinguishes between equality and equivalence of polynomials. As a consequence, a field ideal (ideal consisting of field equations) needs to be used explicitly to reduce the degrees. This has an undesired effect that the polynomials with higher degrees gets constructed and are reduced only later. With proper implementation, the construction of the bigger polynomials could be avoided. It should be mentioned here that SageMath provides quotient rings (a polynomial ring modulo some ideal) which seems to be what we wanted. However, experiments showed that it is not optimized for polynomial ring over finite field modulo the field ideal. So it still computes the larger polynomial and only after that, it reduces it by the ideal. The only difference is, this is done implicitly, not explicitly. As we tried to improve the algorithm, we found we sometimes the field equations for some variables can be improved, so the degree of that variable could be reduce further. Therefore, we decided not to use the quotient ring but to maintain the *reduction ideal* (ideal containing the reduction equations, originally the field equations) explicitly.

Data structures to hold the information about the clauses were created with respect to the expected input format. The class `TPTPParser` is a simple parser for input files from TPTP library in the CNF format. The following classes, with self-explanatory names, reflects the structure of the input and serve to store the input information: `Clause`, `Literal`, `Equality`, `Predicate`, `Term`, `Function` and `Variable`. Input file is parsed into a list a of `Clauses`, each `Clause` holds a list of `Literals`. `Literal` is either `Equality` or `Predicate` and can be positive or negative. `Terms` are arguments of `Literals`. `Equality` has always two terms, `Predicate` has a list of `Terms`, the length depending on the arity of `Predicate`. `Term` can be either `Function` or `Variable`, where `Variable` is defined by its name, and `Function` consists again of list of `Terms`, depending on its arity. The input of the algorithm (as seen in Algorithm 3) is a theory represented as a list of `Clauses`.

The main work is done in `Manager` class. This is the class that holds the current context of the algorithm (current field, polynomial ring, reduction ideal and others) and provides functions for the translation of the clauses and computing the solution of the equations. Much of its work consists of manipulating with polynomials, which is quite easy thanks to functionality provided by SageMath.

There are some options that can alter the run of the algorithm. For setting these options, a configuration file is used and `Configuration` class parses the file and provides the chosen options to the algorithm.

3.3.2 Modifications to the basic algorithm

Since we tweaked the implementation of the algorithm to test the impact of various modifications, the result differs somewhat from the procedure as described in Section 2.3. We now go through the implementation and discuss the modifications that were made.

During the implementation and testing it happened that a solution for some variable has been found during the translating phase. The solution can be *absolute*, when the exact value for a variable is computed (for example $x = 0$), or *partial*, when the value of the variable is expressed in terms of other variables (for example $x = yz$). Finding such solutions early on can help to simplify the translation process (a prominent example is setting first constant in symmetry reduction to 0) and it means less variables in the resulting system of equations, which in turn reduces time spent in the solver. As a consequence, in the implementation, the instantiation *precedes* translation of the clauses. More precisely, the translation is repeated for every possible evaluation of clause variables and when translating a variable, instead of leaving it as a variable, its value from current evaluation is substituted. This has two advantages. Firstly the huge polynomial $\tau(\varphi)$ for clause φ is never constructed, only the smaller polynomials $tr_{in}(\varphi)$ are. This is desired because it takes longer time and more memory to work with bigger polynomials. Secondly some solution can be found from the first instances (especially if we are substituting 0 for all clause variables) and this can make the polynomials in other instances smaller, hence speeding up the rest of the translation.

As mentioned before, some optimizations proposed by Claessen and Sörensson 2003 can be incorporated into the algebraic algorithm as well. Symmetry reduction is the best of them. Static symmetry reduction was formulated in terms of new instantiated clauses and so they can be easily translated and the result added to the system of equations. However, we can use some of them better. For example, the restriction $a_0 = 0$ for the first constant means we have an absolute solution for the variable representing the constant. This can significantly reduce polynomials for clauses containing this constant. For other constants, we do not have a direct solution, but we do obtain polynomials that reduces the degree of variables for these constant better than the original field equations. For example the restriction for the second constant is $a_1 = 0 \vee a_1 = 1$ which translates to $a_1 \cdot (a_1 - 1)$ and that is $a_1^2 - a_1$. The new restriction for a_1, $a_1^2 - a_1 = 0$, is better than the original one $a_1^q - a_1 = 0$ for $q > 2$. If we are out of constants but did not use all elements in symmetry reduction, we can use function symmetry reduction similarly to constants, by restricting the value in the point $\mathbf{0}$ (all arguments are 0) since $f(\mathbf{0}) = c$ where c is the constant coefficient of polynomial representing f. For example restricting unary function after two constants in $GF(5)$ is represented by clause $f(0) = 0 \vee f(0) = 1 \vee f(0) = 2$ which translates to condition $c^3 + 2c^2 + 2c = 0$, restricting the degree of variable for the constant coefficient to 2 instead of 4. This can be used in the translating phase to keep the polynomials of clauses containing restricted constants and functions smaller. In the implementation, ideal for reducing degrees of variables is kept by `Manager`. At the beginning, it contains field equations, but can be updated, for example with restrictions from symmetry reduction.

The next modification we implemented is called predicate restriction. Or more precisely restricting polynomials representing predicates. In the basic algorithm, when searching for a realization of predicate symbol we look for any function and interpret it as a relation (0 as $\top$ and any other element as $\bot$) only later. But we can be strict and demand that the range of the function is $\{0, 1\}$ (Note that this is a subset of *every* finite field). This has a few consequences. Firstly we obtain better restrictive equations for constant coefficients of polynomials representing predicates (in $GF(q)$ for $q > 2$) as the restriction $R(\mathbf{0}) = 0 \vee R(\mathbf{0}) = 1$ translates to $c_R^2 - c_R = 0$. Secondly for each polynomial representing predicate, we obtain new equations containing only the unknown coefficients of this polynomial. Although the resulting system will be bigger, such equations should help find the solution quicker. Finally, the translation of negation of predicate symbol can be simpler. If we know that $\forall \mathbf{x}\ P_R(\mathbf{x}) = 0 \vee P_R(\mathbf{x}) = 1$ then we can replace the original translation $\tau(\neg\psi) = \tau(\psi)^{q-1} - 1$ with $\tau'(\neg\psi) = 1 - \tau(\psi)$ (or $\tau(\psi) - 1$.) as using this new computation of value of negated predicate we do not violate the condition $\neg R(t_1, \ldots, t_n)^{\mathfrak{F}}[e] = 0$ if and only if $R(t_1, \ldots, t_n)^{\mathfrak{F}}[e] \neq 0$ satisfying $\mathfrak{F} \models \neg R(t_1, \ldots, t_n)[e]$ if and only if $\mathfrak{F} \not\models R(t_1, \ldots, t_n)[e]$. This simplifies the translation of negative predicates in clauses since there is no need to compute the power of large polynomials. Unfortunately, this does not work for negated equality. In that case, the old translation with computing the power of the polynomial must still be applied.

If looking for a model of size that is not prime power, a function restriction follows. Polynomials representing restrictive clauses (as defined in Section 2.3.2) are computed and added to the result.

The translation of clauses is implemented as proposed in Definition 2.1 with the exception that the variables are translated to their current value and this is done for every evaluation of variables in the clause. Also, if a solution for some variable has been found earlier, the solution is immediately used instead of the variable. And the translation of negated predicate is simpler if predicate restriction is turned on. The polynomials appearing in the translation are reduced at certain steps with respect to current reduction ideal and they are also checked for possible solution for some variable.

We also implemented a possibility to restrict the size of representing polynomials by restricting the degree of variables in its monomials. By doing this, the algorithm might miss a model if it exists, but it can be interesting to test, for example if some theories have models which have simple representing polynomials (for example only quadratic). This option can be turned on in the configuration of the algorithm.

After the translation, the system of equations is ready for a solver. In this implementation we use computation of Gröbner basis provided by SageMath via SINGULAR. However, when using the method of Gröbner basis, the field equations (or even better restrictions for some variables in our case) must be added to the system to ensure the solution is computed over the finite field. Computation of Gröbner basis is then called. It is good to remember that a Gröbner basis is not unique, it depends on the chosen term order. And the chosen order often have tremendous impact on the computation time. Generally, orders where the total degree of the monomial is taken into account, like degree lexicographic or degree reverse lexicographic, are recommended. However, for the computation of a solution for the system of equations, the lexicographic order is the best, since then the basis has triangular shape and the solution can be read easily. The recommended way is to compute the basis for some of the faster order and then transform the basis to lexicographic one. This is often much faster than computing the lexicographic basis directly and SageMath provides a function for the transformation. If the result is the trivial basis (basis with only the trivial polynomial 1), the system does not have any solution over the finite field and this means the theory does not have a finite model of given size. The algorithm continues by trying the next size. If a non-trivial basis is returned, it encodes all possible solutions of the original system (assignment of variables is a solution to the original system if and only if every polynomial reduces to 0 under this assignment). It is transformed to lexicographic basis for better extraction of some solution. We use a simple algorithm of trying all elements of the field for the extraction of some solution. Fortunately, this is fast for lexicographic basis since it has triangular form. At the end, a model is displayed in the form of representing polynomials using the computed coefficients.

3.4 Testing and results

W tested the algebraic algorithm for FMF mostly on problems from TPTP library and also on some hand-made inputs, e.g. axioms for Boolean algebra, group axioms and axioms for non-Abelian group. The correctness of the algorithm was not disproved by any example we tried. This means that either it correctly returned the smallest model possible or, if it did not finished (due to lack of time or memory), it correctly refuted the sizes it was able to examine.

However, during testing, it soon became clear, that the algorithm is extremely resource-demanding. Since every n-ary predicate or function symbol introduces q^n new variables when working in field $GF(q)$, the representing polynomials grow quickly when incrementing the size of the model. This results in huge polynomials when dealing with rich clauses, that is clauses containing many different function and/or predicate symbols. This is often problem even in the translation phase, as polynomials with hundreds of thousands or even millions of monomials even for small model size (2 or 3) are not that rare (see Example 3.1). Not only does the computation of such polynomials takes long time, but also the memory requirements to store these polynomials are enormous. We tried to introduce some modifications to keep the polynomials smaller, but the effect was insignificant.

Example 3.1. Consider the following clause taken from problem `GRP135-1.005` from TPTP library:

$$\begin{aligned}&cnf(product\_total\_function1, axiom,\\&\quad (\sim group\_element(X)\\&\quad |\sim group\_element(Y)\\&\quad |\ product(X, Y, e_1)\\&\quad |\ product(X, Y, e_2)\\&\quad |\ product(X, Y, e_3)\\&\quad |\ product(X, Y, e_4)\\&\quad |\ product(X, Y, e_5))).\end{aligned}$$

Translating this clause in field $GF(3)$, with symmetry reduction applied, yielded polynomial with 5734890 monomials for variable assignment $X := 1$, $Y := 1$, and polynomial with 4014423 monomials for variable assignment $X := -1$, $Y := 1$. Both polynomials contained 34 variables. In $GF(4)$ these polynomials would contain 72 variables and we would not have enough memory to store them.

To get larger results, we ran the algorithm on 100 problems from TPTP library. After setting filters to satisfiable problems in CNF format, with small formulas and clauses, and rating Easy, 509 problems remained. We randomly picked 100 of them and ran the algorithm on them with limits of 300 seconds and 1.5GB of memory on home laptop. In comparison with PARADOX, which was able to solve all these problems within seconds (and mostly in fractions of second), the results of our algorithm were very poor: it solved only 46 problems, ran out memory in 29 cases and the computation did not finish on time in 25 cases. Moreover. with one exception, in all the successful cases, the problem had model of size only 1 or 2. The results also showed that our model finder completed the search for model of size 3 in very few cases. Even when later tried on a problem of non-Abelian group, which has very simple axioms, the algorithm quickly refuted sizes up to 3, but got stuck in the computation of the Gröbner basis for size 4.

3.4.1 Analysis of results

The experiment confirmed the problems of algebraic approach to FMF. The polynomials occurring in the translation simply grows too much too quickly. This is the result of frequent computing of powers of polynomials for nested arguments, computing power in case of negative literal, and multiplying polynomials of all literals in a clause and also of quick growth of representing polynomials (both in terms of arity and field size). On many problems from TPTP library this yields system of equations with hundreds of variables even for small sizes like 3 or 4. Even when the translation completes successfully, the resulting system contains huge polynomials with many variables and the algorithm and solver is not able to compute the solution of such system.

We are aware that the implementation presented here is far from optimal. We believe that the translation could be faster in a more low-level language like C/C++ and especially if keeping the degrees under given boundary were done directly during multiplication/power operation. It could be further improved by implementing sort inference, since that could allow for even better symmetry reduction. However, we came to the conclusion that the problem of large polynomials is an inherent property of the translation. As a consequence, even if the implementation is faster, the memory requirements will not diminish. And the resulting system of equations is thus intractable even for current state-of-the-art solvers.

For a while we considered a different approach. Instead of performing all the operations with polynomials, we would just remember them in a tree-like structure, since the polynomial is build gradually. This is very compact representation; however, it does not allow the use of algebraic methods and thus the only possible way of searching for a solution of a system in such representation is exhaustive search. Such approach resembles SEM-style methods, but does not have an apparent advantage, so we did not continue this way.

Conclusion

In this thesis we have examined an algebraic approach to finite model finding problem. We showed how, for a given model size, a theory is translated to a system of multivariate polynomial equation (this translation is due to Stanovský, the supervisor of this thesis) such that a model of the theory of given size exists if and only if the system of equations has a solution. We proved that finite structure of first order logic (of prime power size) can be represented in a finite field by polynomials over the field, and used it to prove the correctness of the translation. Finally, we implemented this algorithm in an algebraic model finder and tested the implementation on problems from TPTP library.

The tests showed that the polynomials from the translation are too large. They grow too quickly with increasing model size, they require a lot of memory and the solver is often not able to cope with the resulting system of equations. The algebraic model finder can deal only with the smallest sizes (1 and 2, rarely 3) for hard problems (problems with many different predicate or function symbols in the same clauses, or problems with symbols with high arity) and even for relatively simple problems (problems with simple clauses), it does not finish on sizes starting at 4 or 5. We believe it would be possible to speed up the translation process by better implementation, but the size of resulting polynomials cannot be decreased by much any further. We have thus concluded that although it shows very interesting connection between logic and finite fields, the algebraic approach cannot compete with current state-of-the-art model finders.

Bibliography

Bard, Gregory V. (2009). *Algebraic Cryptanalysis.* 1st. Springer Publishing Company, Incorporated. ISBN: 0387887563, 9780387887562.

Bard, Gregory V., Nicolas T. Courtois, and Chris Jefferson (2007). *Efficient Methods for Conversion and Solution of Sparse Systems of Low-Degree Multivariate Polynomials over GF(2) via SAT-Solvers.* Cryptology ePrint Archive, Report 2007/024. `http://eprint.iacr.org/`.

Baumgartner, Peter et al. (2007). "Computing finite models by reduction to function-free clause logic". In: *Journal of Applied Logic* 2007.

Buchberger, Bruno (2006). "Bruno Buchberger's PhD thesis 1965: An algorithm for finding the basis elements of the residue class ring of a zero dimensional polynomial ideal". Trans. by Michael P. Abramson. In: *Journal of Symbolic Computation* 41.3–4. Logic, Mathematics and Computer Science: Interactions in honor of Bruno Buchberger (60th birthday), pp. 475–511. ISSN: 0747-7171. DOI: `http://dx.doi.org/10.1016/j.jsc.2005.09.007`. URL: `http://www.sciencedirect.com/science/article/pii/S0747717105001483`.

Buchberger, Bruno and Manule Kauers (2010). "Groebner basis". In: *Scholarpedia* 5.10. revision #128998, p. 7763.

Caferra, Ricardo, Alexander Leitsch, and Nicolas Peltier (2004). *Automated Model Building.* Vol. 31. Applied Logic Series. Springer.

Claessen, Koen and Niklas Sörensson (2003). "New Techniques that Improve MACE-style Finite Model Finding". In: *Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications.*

Courtois, Nicolas (2001). "The security of cryptographic primitives based on multivariate algebraic problemss: MQ, MinRank, IP, HFE". Available at `http://www.nicolascourtois.com/papers/phd.pdf`. PhD thesis. Université de Paris 6 - Pierre et Marie Curie.

Courtois, Nicolas et al. (2000). "Efficient Algorithms for Solving Overdefined Systems of Multivariate Polynomial Equations". In: *Proceedings of the 19th International Conference on Theory and Application of Cryptographic Techniques.* EUROCRYPT'00. Bruges, Belgium: Springer-Verlag, pp. 392–407. ISBN: 3-540-67517-5. URL: `http://dl.acm.org/citation.cfm?id=1756169.1756206`.

Decker, Wolfram et al. (2015). SINGULAR *4-0-2 — A computer algebra system for polynomial computations.* `http://www.singular.uni-kl.de`.

Gebser, Martin, Orkunt Sabuncu, and Torsten Schaub (2011). "Finite Model Computation via Answer Set Programming". In: *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Three.* IJCAI'11. Barcelona, Catalonia, Spain: AAAI Press, pp. 2626–2631. ISBN: 978-1-57735-515-1. DOI: `10.5591/978-1-57735-516-8/IJCAI11-437`. URL: `http://dx.doi.org/10.5591/978-1-57735-516-8/IJCAI11-437`.

McCune, William (1994). *A Davis-Putnam Program and its Application to Finite First-Order Model Search: Quasigroup Existence Problems.* Tech. rep.

Stein, W. A. et al. (2015). *Sage Mathematics Software (Version 6.7).* `http://www.sagemath.org`. The Sage Development Team.

Stanovský, D. and L. Barto (2011). *Počítačová algebra.* Matfyzpress. ISBN: 978-80-7378-167-5.

Sutcliffe, G. (2009). “The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0”. In: *Journal of Automated Reasoning* 43.4, pp. 337–362. URL: `http://www.cs.miami.edu/~tptp/`.

Sutcliffe, G. and C. Suttner (2006). “The State of CASC”. In: *AI Communications* 19.1, pp. 35–48.

Zhang, Jian and Hantao Zhang (1995). “SEM: A System for Enumerating Models”. In: *Department of Philosophy University of Wisconsin-Madison Mathematics and Computer Science*, pp. 298–303.

Attachment A - Content of CD

The enclosed CD contains source files, user guide and the electronic version of this thesis.