УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ САДУ

Aleksandar Pejović

Parallel software system for counting finite models

DOCTORAL DISSERTATION

Александар Пејовић

Паралелни програмски систем за пребројавање коначних структура

ДОКТОРСКА ДИСЕРТАЦИЈА

Нови Сад, 2019.

УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, **РБР**:	
Идентификациони број, **ИБР**:	
Тип документације, **ТД**:	монографска документација
Тип записа, **ТЗ**:	текстуални штампани запис
Врста рада, **ВР**:	докторска дисертација
Аутор, **АУ**:	Александар Пејовић
Ментор, **МН**:	др Силвиа Гилезан, редовни професор
Наслов рада, **НР**:	Паралелни програмски систем за пребројавање коначних структура
Језик публикације, **ЈП**:	енглески
Језик извода, **ЈИ**:	енглески, српски
Земља публиковања, **ЗП**:	Република Србија
Уже географско подручје, **УГП**:	
Година, **ГО**:	2019
Издавач, **ИЗ**:	Факултет техничких наука
Место и адреса, **МА**:	Нови Сад, Трг Доситеја Обрадовића 6
Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога)	6/130/42/2/4/0/1
Научна област, **НО**:	математичке науке
Научна дисциплина, **НД**:	логика у рачунарству
Предметна одредница/Кqучне речи, **ПО**:	теорија модела, слободне алгебре, коначна комбинаторика, декларативно програмирање, паралелно програмирање
УДК	
Чува се, **ЧУ**:	Библиотека Факултета техничких наука у Новом Саду
Важна напомена, **ВН**:	
Извод, **ИЗ**:	Ова дисертација се бави развојем паралелног софтверског система за представљање и решавање проблема теорије коначних модела и његовом применом. Износи се теоријска основа система, као и детаљно објашњење имплементације у Пајтону. Конкретно, развијен је паралелан метод за рачунање Булових израза заснован на особинама коначних слободних Булових алгебри. Такође се показује како се различити коначни комбинаторни објекти могу кодирати у формализму Булових алгебри и избројати применом овог поступка. Конкретно, користећи транслацију предикатских формула првог реда на исказне формуле развили смо технику за конструисање и бројање коначних модела теорија првог реда. На крају, развили смо неке опште технике које омогућавају ефективније коришћење нашег система. Ове технике приказујемо на два примера. Први се бави парцијалним уређењима, а други се односи на случајне графове.
Датум прихватања теме, **ДП**:	13.04.2017.
Датум одбране, **ДО**:	

Чланови комисије, **КО**:	Председник:	др Татјана Давидовић, научни саветник	
	Члан:	др Милош Стојаковић, редовни професор	
	Члан:	др Мирослав Поповић, редовни професор	Потпис ментора
	Члан:	др Јелена Иветић, доцент	
	Члан, ментор:	др Силвиа Гилезан, редовни професор	

UNIVERSITY OF NOVI SAD ● **FACULTY OF TECHNICAL SCIENCES**
21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, **ANO**:	
Identification number, **INO**:	
Document type, **DT**:	monographic publication
Type of record, **TR**:	textual printed material
Contents code, **CC**:	PhD thesis
Author, **AU**:	Aleksandar Pejović
Mentor, **MN**:	Dr Silvia Gilezan, full professor
Title, **TI**:	Parallel software system for counting finite models
Language of text, **LT**:	English
Language of abstract, **LA**:	English, Serbian
Country of publication, **CP**:	Republic of Serbia
Locality of publication, **LP**:	
Publication year, **PY**:	2019
Publisher, **PB**:	Faculty of Technical Sciences
Publication place, **PP**:	Novi Sad, Trg Dositeja Obradovića 6
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)	6/130/42/2/4/0/1
Scientific field, **SF**:	mathematical sciences
Scientific discipline, **SD**:	logic in computer science
Subject/Key words, **S/KW**:	model theory, free algebras, finite combinatorics, declarative programming, parallel programming
UC	
Holding data, **HD**:	Library of the Faculty of Technical Sciences in Novi Sad
Note, **N**:	
Abstract, **AB**:	This dissertation is about the development of a parallel software system for representing and solving problems of finite model theory and its application. The theoretical foundation of the system is presented, as well as an in-depth explanation of the implementation in Python. In particular, a parallel method for computing Boolean expressions based on the properties of finite free Boolean algebras is developed. It is also shown how various finite combinatorial objects can be coded in the formalism of Boolean algebras and counted by this procedure. Specifically, using a translation of first order predicate formulas to propositional formulas, we developed a technique for constructing and counting finite models of first order theories. Finally, we have developed some general techniques that enable more effective use of our system. We illustrate these techniques on two examples. The first one deals with partial orders, while the other one is about random graphs.
Accepted by the Scientific Board on, **ASB**:	13.04.2017.
Defended on, **DE**:	

Defended Board, **DB**:		
	President:	Dr Tatjana Davidović, research professor
	Member:	Dr Miloš Stojaković, full professor
	Member:	Dr Miroslav Popović, full professor
	Member:	Dr Jelena Ivetić, assistant professor
	Member, Mentor:	Dr Silvia Gilezan, full professor

Menthor's sign

To Mima.

Acknowledgements

I wish to express my deepest gratitude to my supervisor Dr Žarko Mijajlović, Professor at the Faculty of Mathematics in Belgrade for his valuable advice, insightful suggestions and constant encouragement and support throughout the entire period of my doctoral studies.

I am also very grateful to Dr Silvia Gilezan, Professor at the Faculty of Technical Sciences in Novi Sad, who assumed the supervision of this dissertation after Professor Mijajlović's retirement. Professor Gilezan provided further guidance and offered helpful comments and advice in the final stages of this work.

Abstract

This dissertation is about the development of a parallel software system for representing and solving problems of finite model theory and its application.

We begin by introducing the necessary theory, and then proceed to the description of our system. We give the theoretical foundation of the system, as well as an in-depth explanation of the implementation in Python. In particular, a parallel method for computing Boolean expressions based on the properties of finite free Boolean algebras is developed. Furthermore, it is shown how various finite combinatorial objects can be codded in the formalism of Boolean algebras and counted by this procedure. In other words, using the translation of first order predicate formulas to propositional formulas, we developed a technique for constructing and counting finite models of first order theories.

Finally, we also developed some general techniques that enable more effective use of our system. We illustrate these techniques by two examples. The first one deals with partial orders, while the other one is about random graphs.

Rezime

Ova disertacija se bavi razvojem paralelnog softverskog sistema za predstavljanje i rešavanje problema teorije konačnih modela i njegovom primenom. Iznosi se teorijska osnova sistema, kao i detaljno objašnjenje implementacije u Pajtonu. Konkretno, razvijen je paralelan metod za računanje Bulovih izraza zasnovan na osobinama konačnih slobodnih Bulovih algebri. Takođe se pokazuje kako se različiti konačni kombinatorni objekti mogu kodirati u formalizmu Bulovih algebri i izbrojati primenom ovog postupka. Konkretno, koristeći translaciju predikatskih formula prvog reda na iskazne formule razvili smo tehniku za konstruisanje i brojanje konačnih modela teorija prvog reda. Na kraju, razvili smo neke opšte tehnike koje omogućavaju efektivnije korišćenje našeg sistema. Ove tehnike prikazujemo na dva primera. Prvi se bavi parcijalnim uređenjima, a drugi se odnosi na slučajne grafove.

Teza sadrži uvod i pet poglavlja: *Teorija modela*, *Algebarske i kombinatorne konstrukcije*, *Paralelno brojanje modela*, *Primene* i *Zaključak*. Bibliografija ima 42 referencu. Na kraju teze nalazi se jedan dodatak koji sadrži listinge (izvorni kod) koji se odnose na glavne primere.

Uvod daje jedan pregled tema teorijskog računarstva, teorije konačnih modela i apstraktne algebre koje su u vezi teze. Deluje da je ta veza dvostruka i uzajamna. Jedan aspekt ove veze je u primenama rezultata i metoda pomenutih matematičkih teorija u razvijanju algoritama i programskih principa na kojima se zasniva naš softverski sistem. Uloga slobodnih algebri je naročito naglašena, konkretno slobodnih Bulovih algebri. S druge strane, objašnjava se da bi naš sistem mogao biti efikasno sredstvo u rešavanju kombinatornih problema i zapravo velikog broja matematičkih problema konačnog karaktera koji se mogu izraziti pomoću logike prvog reda. Primena ide od rešavanja problema iz teorije grafova i parcijalno uređenih skupova do određivanja broja konačnih modela proizvoljnih teorija prvog reda. Sa te tačke gledišta naš sistem je uporediv i konkurentan sa sličnim generatorima konačnih modela kao što je Mace4 (razvio ga Bill McCune).

Teorija modela

Svrha ovog poglavlja je samo da utvrdi označavanje, pojmove, terminologiju i navede neke teoreme koje se koriste u tezi. Prema tome, ovo poglavlje ne sadrži originalne ili nove rezultate.

U uvodnom odeljku objašnjava se šta je predmet ove discipline matematičke logike. Neformalno govoreći uzima se da je teorija modela = univerzalna algebra + logika.

U odeljcima *Jezici prvog reda* i *Termi i formule* uvode se osnovne sintaksne konstrukcije za predikatski račun prvog reda: data je formalna definicija jezika prvog reda, promenljivih i induktivne definicije terma i formula.

U odeljku *Teorije prvog reda* daju se logičke aksiome, zatim definicija aksioma teorija prvog reda, pravila dedukcije i precizna definicija teorema teorija prvog reda. Takođe se navode neka osnovna tvrđenja sintaksnog karaktera o teorijama prvog reda, kao što su lema o novoj konstanti i teorema dedukcije. Ukratko je raspravljeno i pitanje odlučivosti aksiomatskih teorija prvog reda, u sklopu čega je dat jedan dokaz teoreme koja kaže da ako je teorija sa rekurzivnim skupom aksioma kompletna, onda je i odlučiva.

Naveli smo sledeće primere:

1. Čist predikatski račun prvog reda (algebarsko-relacijski jezik je prazan skup).

2. Razne vrste uređenja.

3. Bulove algebre.

U odeljku *Modeli* data je stroga definicija modela, tj. relacijsko-operacijske strukture, podmodela, homomorfizama, izomorfizama i automorfizama, dok je u odeljku *Relacija zadovoljenja* opisana induktivna definicija relacije zadovoljenja.

Mada na prvi pogled odeljak *Metoda novih konstanti* nema direktne veze sa kasnijim istraživanjem u tezi, metodološki je ipak povezana sa postupkom kodiranja predikatskih formula indeksiranim iskaznim promenljivama. Naime, daje se jedna zanimljiva konstrukcija kojom se pokazuje da se relacija zadovoljena može uvesti, uz uvođenje novih simbola konstanti, samo za rečenice, umesto na širem skupu, tj. skupu svih formula datog jezika prvog reda. U našem slučaju, prilikom svođenja predikatskih aksioma na iskazne formule, simboli konstanata su zapravo imena elemenata konačnog domena tražene strukture.

Na kraju, istaknimo da su uvedeni formalizmi neophodni u delu teze koji se odnosi na implementaciju softvera. To je posledica činjenice da se teza odnosi upravo na generisanje modela nad konačnim domenima teorija prvog reda.

Algebarske i kombinatorne konstrukcije

U ovom poglavlju se obrađuju dve teme algebarske i kombinatorne prirode.

Grupovno dejstvo na model U našem radu važnu ulogu će imati pojam grupovnog dejstva na model. Više koncepata se dovode u vezu sa grupovnim dejstvom, kao što su orbite, stabilizatori,

klasovna jednakost, itd. Na primer, mi možemo da merimo simetriju neke strukture brojem orbita. U ekstremnom slučaju imamo strukturu visokog stepena simetrije sa samo jednom orbitom. Takve strukture imaju tranzitivno grupovno dejstvo. Videćemo da je na neki način simetrija strukture inverzna mogućnosti definisanja u strukturi.

Od važnijih teorema iz ove oblasti predstavljena je i dokazana klasovna jednakost za grupovno dejstvo.

Formule inverzije Druga tema se odnosi na tzv. formule inverzije, što je važna grana kombinatorike. Ovu teoriju koristimo za pronalaženje naročitih tipova automorfizama modela i permutacija njihovih domena, na primer sa određenim brojem fiksnih tačaka. Ovaj odeljak sadrži originalan doprinos, a to je jedno proširenje Guldovih formula inverzije [16], [17].

Najpre se pokazuje na primeru za $m = 2$, korišćenjem Hophovih algebri i svojstava Čebiševljevih polinoma, da važi dobro poznata formula inverzije

$$g_n = \sum_k \binom{n}{k} f_{n-2k} \;\Leftrightarrow\; f_n = \sum_{2k \leqslant n} (-1)^k \frac{n}{n-k} \binom{n-k}{k} g_{n-2k}, \quad f, g \in \mathcal{F}.$$

Zatim se izvodi Guldovo uopštenje

$$g_n = \sum_k \binom{n}{k} f_{n-mk} \;\Leftrightarrow\; f_n = \sum_{0 \leqslant k \leqslant \frac{n}{m}} (-1)^k \frac{n}{n-rk} \binom{n-rk}{k} g_{n-mk},$$

ali i naša dalja generalizacija koja predstavlja originalan prilog za rekurentnu formulu

$$g_n = \sum_k \binom{n}{k} f_{n-mk+l}, \quad 0 \leqslant l < m.$$

Verujemo da je naš pristup nov pošto se zasniva na Hophovim algebrama i linearnom funkcionalu $\theta = \pi + \pi^{-r}$ na prostoru C^Z, π je funkcija projekcije, C je skup kompleksnih brojeva, a Z je skup celih brojeva. Većina rezultata iz ovog odeljka je objavljena u radu [31].

Paralelno brojanje modela

Ovo je najveće i ujedno centralno poglavlje u disertaciji. Verujemo da je većina rezultata i programskih rešenja ovde originalna. Sastoji se od četiri odeljka.

Uvod U prvom odeljku se detaljno obrađuje problem nalaženja spektra (broja modela) konačnih modela teorija prvog reda i na koji način je to povezano sa rešavanjem drugih kombinatornih problema na primer u teoriji grafova ili teoriji particija. Objašnjeno je da je

osnovna ideja sadržana u zapažanju da je

$$f(b_1, b_2, \dots, b_n) \text{ bulovski vektor koji kodira punu DNF za } f,$$

gde je $f(x_1, x_2, \dots, x_n)$ bulovski term po promenljivama $x_1, x_2, \dots, x_n$, dok su $b_1, b_2, \dots, b_n$ slobodni bulovski vektori (generatori) slobodne Bulove algebre $\mathbf{\Omega}_n$ koju generišu. Ovo omogućava proveru logičke validnosti za $f(x_1, x_2, \dots, x_n)$, tretirajući je sada kao iskaznu formulu. Opštije rečeno, za f mogu da se izračunaju modeli μ, tj. valuacije takve da je $f[\mu] = 1$. Pošto je $\mathbf{\Omega}_n \cong \mathbf{2}^{2^n}$, sledi da se vrednost $f^{\mathbf{\Omega}_n}(b_1, b_2, \dots, b_n)$ može izračunati koristeći paralelne mogućnosti logičkih operacija nad bitovima u savremenim računarskim uređajima kao što su CPU i GPU. Ova ideja se javila još 1998. u Mijajlović [30], ali je u potpunosti ispitana i sprovedena u delo kao softverski sistem u ovoj tezi.

Promenljive Drugi odeljak daje detaljnu algebarsku pozadinu u vezi sa konačnim slobodnim Bulovim algebrama, ali i slobodnim algebrama uopšte.

Interpretacija promenljivih Smatramo da je skup promenljivih bilo kakav neprazan skup V tako da nijedno $v \in V$ nije konačan niz drugih elemenata iz V. Ova pretpostavka obezbeđuje jedinstvenu čitljivost terma i formula. Dalje, usvojimo da je I skup valuacija domena A, tj. $I = \{\mu \mid \mu\colon V \to A\}$.

Definicija (Interpretacija promenljivih)**.** Neka je v promenljiva iz V. Interpretacija promenljive v u domenu A je preslikavanje $\hat{v}\colon I \to A$ definisano kao $\hat{v}(\mu) = \mu(v)$, $\mu \in I$.

Pojam interpretacije promenljivih će igrati fundamentalnu ulogu u našoj analizi i implementaciji programa, ali može biti od koristi i u drugim slučajevima. Na primer, koristeći ovaj pojam lako se dokazuje čuvena Birkhofova teorema o postojanju slobodnih algebri i druge teoreme u vezi sa njom, kao što je Bikhofova HSP teorema.

Novina ovog dokaza u odnosu na standardni dokaz je da se ne koristi pojam term algebre (apsolutno slobodne algebre) i da ima modelsko-teoretsku prirodu.

Slobodni bulovski vektori Dobro je poznato da su algebre $\mathbf{2}^{2^n}$ konačne slobodne Bulove algebre sa n slobodnih generatora. Struktura i osobine slobodnih bulovskih vektora $\mathbf{\Omega}_n = \mathbf{2}^{2^n}$ se detaljno razmatraju u [30].

Bavićemo se naročito slobodnim generatorima $\mathbf{\Omega}_n$ sledećeg oblika. Neka su a_i, $i = 0, 1, \dots, 2^n - 1$, binarni razvoji celih brojeva dopunjeni vodećim nulama do dužine n. Neka je M matrica čije su kolone vektori a_i. Kao što se konstatuje u [30], binarni vektori b_i, $i = 1, 2, \dots, n$,

koji obrazuju vrste M su slobodni vektori $\mathbf{\Omega}_n$. U slučaju $n = 3$ matrica M i vektori b_i su

$$M = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix},$$

$b_1 = 00001111$, $b_2 = 00110011$, $b_3 = 01010101$.

Računanje bulovskih izraza Opisujemo ukratko paralelni algoritam za računanje $d = t^{\mathbf{\Omega}_n}(b_1, \ldots, b_n)$. Pretpostavimo da raspolažemo sa 2^k-bitnim procesorom, $k < n$. Svaki od vektora b_i se deli na 2^{n-k} uzastopnih sekvenci. Sledi da se b_i sastoji od 2^{n-k} blokova b_{ij} veličine 2^k. Da bi se našlo d, računaju se blokovi $d_j = t(b_{1j}, \ldots, b_{nj})$ veličine 2^k koristeći operacije nad bitovima za $j = 1, 2, \ldots, 2^{n-k}$. Tada je kombinovani vektor $d_1 d_2 \ldots d_{2^{n-k}}$ upravo izlazni vektor d. Ukupno vreme da se izračuna d približno iznosi $T = 2^{l+n-k}\delta$, gde je 2^l ukupan broj čvorova u binarnom stablu izraza koji odgovara termu t, a δ je vremenski interval za računanje jedne logičke operacije nad bitovima[1].

Računanje konačnih modela Treći odeljak se uglavnom bavi sintaksnim transformacijama i algoritmima za prevođenje predikatskih formula prvog reda φ u iskazne formule φ^* na konačnom domenu A.

Koristeći prevođenje iz predikatske logike prvog reda $L_{\omega\omega}$ u iskaznu logiku L_ω možemo da formulišemo i računski rešimo različite probleme na konačnim strukturama.

Prevođenje iz $L_{\omega\omega}$ u L_ω Metoda za kodiranje nekih pojmova, uglavnom kombinatorne prirode koji se odnose na prebrojive strukture prvog reda, pomoću teorija iskaznog računa L_{ω_1} predstavljena je u [33]. Tamošnji glavni cilj bio je da se prouči kompleksnost ovih pojmova u Borelovoj hijerarhiji. Kodiranje je tamo dato preslikavanjem $*$ koje smo ovde prilagodili za naše potrebe.

Neka je L konačan jezik prvog reda, a L_A jezik L proširen simbolima konstanti – imenima elemenata konačnog domena A. Skup iskaznih slova $\mathcal{P}$ definišemo na sledeći način

$$\begin{aligned} \mathcal{P} \;=\; & \{p_{Fa_1\ldots a_k b} \mid a_1, \ldots, a_k, b \in A, F \text{ je } k\text{-arni funkcijski simbol } L\} \cup \\ & \{q_{Ra_1\ldots a_k} \mid a_1, \ldots, a_k \in A, R \text{ je } k\text{-arni relacijski simbol } L\}. \end{aligned} \tag{1}$$

Zatim rekurzivno definišemo preslikavanje prevođenja $*$ iz skupa $Sent_{L_A}$ svih $L_{\omega\omega}$-rečenica L_A u skup $L_\omega^{\mathcal{P}}$ iskaznih formula nad $\mathcal{P}$. U osnovi, preslikavanje $*$ prevodi univerzalne i

[1] Kod savremenih računara $\delta \approx 10^{-9}$ sekundi

egzistencijalne kvantifikatore redom u konačne konjunkcije i disjunkcije parametrizovanih formula bez kvantifikatora.

Na primer, ako je φ rečenica kojom se iskazuje asocijativnost binarnog funkcijskog simbola $\cdot$, biće da je rezultat primene $*$ na $i \cdot j = u$ upravo p_{iju} i da je φ^* na domenu $I_n = \{0, 1, \ldots, n-1\}$ ekvivalentno sa

$$\bigwedge_{i,j,k,u,v,l<n} ((p_{iju} \wedge p_{jkv} \wedge p_{ukl}) \Rightarrow p_{ivl})$$

Korespondencija između modela T i T^* Koristeći prevođenje $*$ dajemo metodu za konstruisanje i brojanje konačnih modela teorija prvog reda konačnog jezika L.

Sa $\mathscr{L}_{T,n}$ obeležićemo skup svih označenih modela T veličine n. Pretpostavimo da je T konačna teorija. Neka je $\mathcal{P}$ skup iskaznih slova definisanih sa (1) za $A = I_n$ i jezik L, i neka je $T^* = \{\varphi^* | \varphi \in T\}$. Dalje, neka $\mathfrak{M}(T^*) \subseteq 2^{\mathcal{P}}$ označava skup svih modela T^*, tj. valuacija koje zadovoljavaju sve iskazne formule u T^*. Sledeća konstrukcija opisuje korespondenciju između označenih modela T i modela T^*.

Funkcija h koja dodeljuje svakoj $\mu \in \mathfrak{M}(T^*)$ jedan označen model $h(\mu) = \mathbf{A}$ teorije T definiše se na sledeći način. Neka $a_1, \ldots, a_k, b \in I_n$.

Ako je $F \in L$ k-mesni funkcijski simbol, tada

$$F^{\mathbf{A}}(a_1, \ldots, a_k) = b \quad \text{iff} \quad \mu(p_{Fa_1 \ldots a_k b}) = 1.$$

Ako je $R \in L$ k-mesni relacijski simbol, tada

$$\mathbf{A} \models R[a_1, \ldots, a_k] \quad \text{iff} \quad \mu(q_{Ra_1 \ldots a_k}) = 1. \tag{2}$$

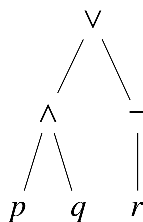
Indukcijom po kompleksnosti formule φ dokazali smo:

Teorema. *Preslikavanje h kodira modele iz $\mathscr{L}_{T,n}$ modelima T^*.*

Ova teorema je naša polazna tačka u određivanju konačnih modela T veličine n. Naime, pošto je T^* konačna, aksiome T možemo zameniti s jednom iskaznom formulom $\theta = \bigwedge_{\psi \in T^*} \psi$.

Softverska implementacija U četvrtom odeljku se detaljno objašnjavaju implementacija našeg sistema i njegove glavne komponente.

Kada smo započeli razvoj sistema, naša glavna motivacija bio je algoritam opisan u pododeljku *Računanje bulovskih izraza*. Stoga ne treba da iznenadi da je upravo paralelno brojanje modela njegova glavna karakteristika. Međutim, to je tek samo jedan aspekt sveukupnog sistema. Pored toga, razvili smo i funkcionalnosti kao što su: domensko-specifični jezik (DSL)



Sl. 1 AST za $p \wedge q \vee \neg r$

za formulisanje teorija prvog reda, njihovo prevođenje u iskazni račun i parcijalna evaluacija iskaznih formula da pomenemo nekoliko istaknutih.

Celokupni sistem je implementiran u programskom jeziku Pajton sa izuzetkom hardverski ubrzanih primitiva, poput primitive koja se koristi za paralelno brojanje modela, a koje su implementirane korišćenjem OpenCL-a, platforme za paralelno računanje.

Sistem smo osmislili kao svojevrsni set alata za kreiranje brojača modela koji su prilagođeni datom problemu, ili drugim rečima, kao biblioteku za Pajton koja obezbeđuje sve neophodne gradivne blokove za njihovu konstrukciju. Ideja je da se omogući da se namenski brojači modela razvijaju na isti način kao i bilo koji drugi program u Pajtonu. Stoga je važan cilj implementacije bio da se osigura da funkcionalnost koju pruža sistem bude dobro integrisana sa Pajtonom.

U ostatku odeljka proći ćemo kroz detalje implementacije svake važne funkcionalnosti sistema.

Reprezentacija teorija prvog reda Iskazne formule zauzimaju centralno mesto unutar sistema. Ima više razloga za to. Najvažniji je taj da se OpenCL C kod za hardversko ubrzanje generiše na osnovu iskaznih formula. Zbog toga nam je bila potrebna struktura podataka za reprezentaciju iskaznih formula koja podržava neposredno generisanje koda. Jedna takva struktura naziva se apstraktno sintaksno stablo ili AST. U svojstvu primera, Sl. 1 prikazuje AST sledeće iskazne formule

$$p \wedge q \vee \neg r. \tag{3}$$

Da bismo gradili AST-ove terma jezika L, potreban nam je poseban tip AST čvora za svaki funkcijski simbol F, kao i jedan za konstante i jedan za promenljive. Svaki tip čvora implementiran je kao zasebna podklasa apstraktne klase `Term` koja predstavlja terme jezika.

Koristeći uvedene klase moguće je napraviti AST bilo kojeg terma jezika L. Na primer, AST formule (3) može da se napravi pomoću sledećeg Pajtonovog izraza

```
Or(And(Var('p'), Var('q')), Not(Var('r'))).
```

Međutim, pravljenje AST-ova na ovakav način je prilično zamorno (naročito za složenije formule). Štaviše, rezultujući Pajtonovi izrazi nisu lako razumljivi, tj. teško je reći kojoj iskaznoj formuli dati Pajtonov izraz odgovara. Zato smo u cilju pojednostavljenja unosa formula u sistem razvili DSL za pravljenje njihovih AST-ova.

Osnovna ideja koja stoji iza implementacije našeg DSL-a je jednostavna: hoćemo da pišemo iskazne formule kao Pajtonove izraze i da dobijemo odgovarajuće AST-ove kao rezultat njihove evaluacije. Stoga smo naš DSL implementirali kao unutrašnji DSL [14] Pajtona.

Prednost ovog dizajna je što ne zahteva razvoj dodatnog parsera i dobro se integriše sa ostatkom sistema. Na primer, možemo koristiti AST čvorove direktno u Pajtonovim izrazima.

Kao rezultat toga, da bismo napravili AST iskazne formule, dovoljno je da je prepišemo kao ekvivalentan Pajtonov izraz nad bitovima po AST promenljivama (`Var` čvorovi). Drugim rečima, AST formule (3) se pravi na sledeći način

```
p & q | ~r,
```

gde su `p`, `q` i `r` redom `Var('p')`, `Var('q')` i `Var('r')`.

Parcijalna evaluacija iskaznih formula Prilikom generisanja OpenCL C koda, broj promenljivih generisane funkcije odgovara broju slobodnih promenljivih početne iskazne formule. Stoga, ako neke slobodne promenljive imaju konstantne vrednosti, generisani kod neće biti optimalan. Obratno, ako znamo da su vrednosti nekih promenljivih konstantne ili ako smo ih namerno fiksirali, možemo da upotrebimo to znanje za generisanje optimizovanog OpenCL C koda.

Optimizacija funkcija putem specijalizacije kada su neki njihovi ulazi fiksirani na konstantne vrednosti poznata je kao parcijalna evaluacija. Osnovna teorija i metode parcijalne evaluacije u teorijskom računarstvu su dobro objašnjeni u [22]. U našem slučaju, hoćemo da specijalizujemo AST početne iskazne formule. Tačnije, hoćemo da dobijemo novi AST koji odgovara početnoj iskaznoj formuli u kojoj su neke slobodne promenljive zamenjene njihovim konstantnim vrednostima. Štaviše, hoćemo da napravimo redukovani AST, bez logičkih konstanti `true` i `false`, izuzev kada je i sam AST konstanta.

Na primer, pretpostavimo da već imamo izraz `e` koji odgovara formuli (3), to jest `e = Expression(p & q | ~r)`. A sada pretpostavimo da znamo da je `q` uvek netačno. Tada bismo mogli upotrebiti parcijalnu evaluaciju da specijalizujemo `e` za poznate činjenice. Konkretno, izvršavajući

```
s = e.eval({q: false}),
```

dobijamo novi izraz `s` u kojem su sve pojave `q` zamenjene sa `false`. Drugim rečima, izraz `s` je kao da je dobijen na sledeći način

```
s = Expression(p & false | ~r).
```

Ako detaljnije pogledamo formulu koja odgovara specijalizovanom izrazu `s`, očigledno je da se može dodatno pojednostaviti. Naime,

$$p \wedge \bot \vee \neg r \Leftrightarrow \neg r.$$

To jest, možemo da eliminišemo još jednu iskaznu promenljivu: p. Ali da bi se ovo postiglo, neophodno je koristiti simboličko računanje, što je, neiznenađujuće, jedna od tehnika koje se obično koriste za parcijalnu evaluaciju [22].

Simboličko računanje smo realizovali tako što zamenjujemo AST čvorove njihovim pojednostavljenjima kad god je to moguće. Kao rezultat toga, sledeće poređenje je tačno

```
Expression(p & q | ~r).eval({q: false}) == Expression(~r).
```

Hardversko ubrzanje OpenCL platforma modeluje račun kao višedimenzionalni niz niti, gde svaka nit izvršava isti računski kernel[2], ali sa različitim podacima. Naš model podataka se već sasvim lepo uklapa u OpenCL platformu s obzirom da radimo sa segmentima slobodnih Bulovih vektora, tj. segmentima nizova bitova. Svaki računski kernel treba da implementira obradu jednog segmenta.

Sistem koristi više računskih kernela. Po jedan za svaku ubrzanu primitivu koju obezbeđuje. Međutim, svi kerneli dele zajedničku funkciju `f` koja implementira evaluaciju iskazne formule t. Ova funkcija predstavlja suštinu naših računskih kernela.

Implementacija `f` se zasniva na svojstvu aranžiranja slobodnih Bulovih vektora $\omega_0, \ldots, \omega_{n-1}$ koje sledi iz sledećeg izomorfizma algebarskih struktura:

$$\mathbf{\Omega}_n \simeq \mathbf{\Omega}_k^{2^{n-k}}.$$

Ubrzane primitive Sistem obezbeđuje brojne hardverski ubrzane Pajtonove primitive koje mogu da se koriste pri razvoju namenskih brojača modela. Dostupne su kao metode sledeće klase:

`class` **`ClRunner`**`(device)`

Predstavlja jedinicu za izvršavanje koja koristi hardversko ubrzanje za računanje rezultata dostupnih primitiva.

Svaka jedinica (instanca klase `ClRunner`) ima `device`, stvarni OpenCL računski uređaj koji se koristi za izvršavanje.

[2] U OpenCL platformi računski kernel je funkcija deklarisana u OpenCL programu koja se izvršava na računskom uređaju kao što je CPU ili GPU.

`eval(expression)`
Izračunava sve modele datog izraza `expression` koristeći paralelnu evaluaciju.

`count(expression)`
Izračunava broj modela datog izraza `expression` koristeći paralelno brojanje.

`findmodel(expression)`
Izračunava model datog izraza `expression` koristeći paralelnu pretragu.

`exist(expression)`
Vraća `True` ako dati izraz `expression` ima model, u protivnom `False`.

Kao pogodnost, sistem nudi sledeće predefinisane jedinice za izvršavanje:

`class GPURunner`
Podklasa klase `ClRunner` koja koristi GPU sa najvećim brojem jezgara za izvršavanje.

`class CPURunner`
Podklasa klase `ClRunner` koja koristi CPU za izvršavanje.

Prema tome, da bismo pronašli broj modela formule (3) koristeći GPU, izvršili bismo sledeće

```
GPURunner().count(Expression(p & q | ~r)).
```

Primer Ovaj odeljak zaključujemo jednim ilustrativnim primerom. Klasični 9x9 sudoku je dobro poznat problem, nije težak za razumevanje, a ipak zahteva namenski rešavač da bi se rešio pomoću našeg sistema. Zbog toga je pogodan da se na njemu pokaže kako se različiti delovi sistema međusobno uklapaju i kako celokupna integracija sa Pajtonom radi.

Poređenje sa sličnim softverom Na kraju ovog poglavlja naš sistem se poredi sa drugim softverskim paketima sposobnim za nalaženje konačnih modela kao što su Mace4, Paradox i PSATO.

Primene

Ovo poglavlje sadrži dva netrivijalna primera upotrebe našeg sistema.

Mreže U narednim pododeljcima razvijamo jednu opštu tehniku za smanjenje broja iskaznih promenljivih korišćenjem simetrija svojstvenih problemu koji se razmatra. Formalno govoreći, svodimo je na problem određivanja orbita pod dejstvom grupe automorfizama konačne strukture na svom domenu. Ove tehnike i algoritme ilustrujemo pisanjem programa u našem sistemu za jedan prilično opšti problem brojanja posebnih vrsta parcijalnih uređenja i mreža.

Uklanjanje promenljivih Pretpostavimo da neka teorija T opisuje klasu konačnih modela. Skup iskaznih slova $\mathcal{P}$ definisan sa (1) i koji se pojavljuje u prevodu iz T u T^* velik je čak i za malo n za domene $A = I_n$ iz kojih se $\mathcal{P}$ generiše. Stoga, nama je potreban način da eliminišemo neke iskazne promenljive koje se pojavljuju u T^*. Bilo kakav postupak eliminacije promenljivih iz $\mathcal{P}$ zvaćemo uklanjanje promenljivih.

Neka je T konačna teorija jezika L i pretpostavimo da su $\varphi_0(x), \ldots, \varphi_{k-1}(x)$ formule L za koje T dokazuje da su međusobno razdvojene, tj. za $i \neq j$, $T \vdash \neg\exists x(\varphi_i(x) \wedge \varphi_j(x))$. Dalje, pretpostavimo da one definišu konstante u T, drugim rečima za svako i

$$T \vdash \exists x(\varphi_i(x) \wedge \forall y(\varphi_i(y) \Rightarrow \varphi_i(x))).$$

Naše korišćenje definabilnih konstanti za uklanjanje promenljivih zasniva se na zapažanju da u mnogim slučajevima teorija T određuje vrednosti atomičnih formula gde figurišu neke od definabilnih konstanti. Kao posledica, odgovarajuća iskazna slova iz $\mathcal{P}$ imaju definitivnu vrednost. Na primer, pretpostavimo da je R dvomesni relacijski simbol i da T dokazuje $\forall x R(c_0, x)$. Tada možemo uzeti $p_{0i} = 1$, $i = 0, \ldots, n-1$. Odnosno, ako je $\mathcal{P}$ generisan po I_n, n iskaznih promenljivih je ubijeno u $\mathcal{P}$. Broj preostalih promenljivih je $n^2 - n$.

Definabilne particije Izložena ideja sa definabilnim konstantama može da se proširi i na definabilne podskupove. Radi jednostavnosti usvojićemo da je $L = \{R\}$, gde je R simbol binarne relacije.

Niz formula $\Delta = \theta_1(x), \ldots, \theta_m(x)$ jezika L zvaćemo definabilnom particijom za T_n ako T_n dokazuje:

1. $\forall x(\theta_1(x) \vee \ldots \vee \theta_m(x))$.
2. $\neg\exists x(\theta_i(x) \wedge \theta_j(x))$, $\quad 1 \leqslant i \leqslant j \leqslant m$.

Reći ćemo da je Δ dobra definabilna particija ako postoje formule $S_{ij}(x, y)$, $1 \leqslant i, j \leqslant m$, takve da je svaka $S_{ij}(x, y)$ jedna od $R(x, y)$, $R(y, x)$, $\neg R(x, y)$, $\neg R(y, x)$ i T_n dokazuje:

$$\forall xy((\theta_i(x) \wedge \theta_j(y)) \Rightarrow S_{ij}(x, y)), \quad 1 \leqslant i \leqslant j \leqslant m.$$

U svakom označenom modelu $\mathbf{A}$ od T_n, Δ određuje niz $\mathcal{X}$ definabilnih podskupova $X_1, \ldots, X_m$. Niz $\mathcal{X}$ sa ovim svojstvom nazivaćemo c-particijom.

Naša ideja za upotrebu dobre definabilne particije Δ za generisanje označenih modela $\mathbf{A}$ od T_n sastoji se u sledećem. Usvajamo da iskazno slovo p_{ij} predstavlja $R^{\mathbf{A}}(i, j)$ kao što je opisano sa (2). Generišemo sve c-particije $\mathcal{X} = (X_1, \ldots, X_m)$ od I_n, uzimajući da X_i odgovara θ_i. Za svaku $\mathcal{X}$ dodeljujemo vrednosti konkretnom p_{ij} na sledeći način. Ako $S(x, y)$ jeste $R(x, y)$,

tada stavljamo $p_{ij} = 1$ za $i \in X_k$ i $j \in X_l$, $k \leqslant l$, a ako $S(x, y)$ jeste $\neg R(x, y)$, stavljamo $p_{ij} = 0$. Na sličan način dodeljujemo vrednosti p_{ji} u zavisnosti da li $S(x, y)$ jeste $R(y, x)$ ili $\neg R(y, x)$. Čineći to, dobijamo iskaznu teoriju $T_{\mathcal{X}} \subseteq T_n^*$ sa smanjenim brojem nepoznatih iz $\mathcal{P}$.

Nesvodljive komponente Pretpostavimo da je $\mathbf{A}$ neki model teorije T_n. Tada svaka nesvodljiva formula θ definiše neki podskup X_θ u $\mathbf{A}$. Ako X_θ nije prazan, onda se ne može razdvojiti na dva neprazna definabilna podskupa bez zajedničkih elemenata, tj. X_θ ne sadrži pravi neprazan definabilan podskup. Za bilo koji neprazan podskup u $\mathbf{A}$ koji je definabilan pomoću neke nesvodljive formule kaže se da je nesvodljiva komponenta $\mathbf{A}$. Međutim, postoji jedan korisniji način za opisivanje nesvodljivih formula i komponenti kao što ćemo pokazati u nastavku.

Neka je $\mathbf{A}$ model teorije T_n za $A = I_n$ i neka je $G = \mathrm{Aut}(\mathbf{A})$ i $H\colon G \to \mathrm{Sym}(A)$ definisano sa $H(g)(x) = g(x)$, $g \in G$, $x \in A$. Tada je H grupovno dejstvo G na A i orbite ovog dejstva su $x^G = \{g(x)\colon g \in G\}$, $x \in A$.

Tvrđenje. *Neka je T konačna teorija jezika L. Tada su nesvodljive komponente u nekom modelu $\mathbf{A}$ od T_n tačno orbite pod dejstvom H.*

Naša opšta ideja za generisanje označenih modela konačnih teorija nekog jezika sa jednim binarnim relacijskim simbolom je da koristimo sledeće tvrđenje, koje smo u tezi dokazali, za uklanjanje promenljivih.

Tvrđenje. *Neka je T konačna teorija $L = \{R\}$, R simbol binarne relacije, a $S(x, y)$ jedna od $R(x, y)$, $R(y, x)$, $\neg R(x, y)$, $\neg R(y, x)$. Usvojimo da je $\theta(x)$ nesvodljiva formula T_n, a $\varphi(x)$ neka formula L. Pretpostavimo*

$$T_n \vdash \exists x(\theta(x) \wedge \forall y(\varphi(y) \Rightarrow S(x, y))).$$

Neka je $\mathbf{A}$ model T_n, X_θ komponenta $\mathbf{A}$ pridružena θ, a X_φ podskup definabilan u $\mathbf{A}$ preko φ. Tada $\mathbf{A} \models S[a, b]$ za svako $a \in X_\theta$ i svako $b \in X_\varphi$.

Brojanje mreža: implementacija programa Ovde opisujemo opštu strukturu programa za brojanje mreža i drugih vrsta parcijalnih uređenja zasnovanog na teoriji razvijenoj u prethodnim pododeljcima.

Turnirski grafovi Drugi primer se odnosi na posebnu vrstu turnira, takozvane Kauerove grafove, i pokazuje kako se 0–1 zakon za konačne modele i slučajne grafove može koristiti u kombinaciji sa našim sistemom da se dokaže da ti grafovi sa n čvorova postoje za svako $n \geqslant 7$.

Kauerovi grafovi su strukture $(T, <)$ koje zadovoljavaju sledeće aksiome:

1. $\forall x \exists y (x < y)$.
2. $\forall x \forall y (x < y \Rightarrow \neg(y < x))$.
3. $\forall x \forall y (x < y \Rightarrow (\exists z (x < z \wedge z < y)))$.

Najpre se dokazuje da Kauerov graf sa najmanjim brojem elemenata ima 7 čvorova. Ispostavlja se da je taj graf turnir, tj. zadovoljava aksiomu: $x < y \vee y < x$. Stoga se u nastavku izučavanja ovog problema ograničavamo na turnire.

Zatim se uz pomoć sistema generišu svi (do na izomorfizam) Kauerovi grafovi od 8 i 9 elemenata. Dalje se dokazuje metodom slučajnih grafova koju je uveo P. Erdös da za svako $n > 21$ postoji Kauerov graf (turnir) sa n čvorova. Najzad, na osnovu ovog rezultata i 0–1 zakona za konačne modele, dokazuje se da verovatnoća da slučajno izabrani turnir bude Kauerov graf teži jedinici kada broj čvorova raste prema beskonačnosti. Drugim rečima za dovoljno veliko n potreban je mali broj pokušaja da se slučajnim generisanjem turnira dobije baš Kauerov graf sa n čvorova. Ispostavlja se da je $n = 10$ taj dovoljno veliki broj. Upravo koristeći ovu činjenicu, lako se generišu Kauerovi turniri čiji je broj čvorova $n = 10, 11, 12, \ldots, 21$. Otuda sledi teorema, da za svako $n \geqslant 7$ postoji Kauerov turnir sa n čvorova.

Zaključak

U poslednjem poglavlju dat je objedinjeni pregled svih originalnih rezultata dobijenih u ovoj tezi. Svakako, najbitniji rezultat teze je razvijeni paralelni softverski sistem za prebrojavanje konačnih struktura. Pored toga, u poglavlju je dat i kratak osvrt na druge softvere sličnog tipa. Na kraju poglavlja predložene su neke mogućnosti daljeg razvoja našeg softvera, a može se raditi i na poboljšanju njegovih performansi.

Table of contents

List of figures xxiii

List of tables xxiv

List of listings xxv

Nomenclature xxvi

1 Introduction 1

2 Model theory 3
2.1 Introduction to model theory 4
2.2 First-order languages 4
2.3 Terms and formulas 5
2.4 First-order theories 9
2.5 Examples of theories 12
2.6 Models 15
2.7 Satisfaction relation 19
2.8 Method of new constants 24

3 Algebraic and combinatorial constructions 27
3.1 Group action on a model 27
3.2 Inversion formulas 29
3.2.1 Hopf algebra of projection functions 30
3.2.2 Linear functional $\theta = \pi + \pi^{-r}$ 31
3.2.3 Gould inversion formula 36
3.2.4 Example 39

4 Parallel model counting 43

4.1 Introduction to model counting 43
4.2 Variables 45
4.2.1 Interpretation of variables 45
4.2.2 Free Boolean vectors 47
4.2.3 Computing Boolean expressions 49
4.3 Computing finite models 49
4.3.1 Translation from $L_{\omega\omega}$ to L_{ω} 50
4.3.2 Correspondence between models of T and T^* 51
4.4 Software implementation 53
4.4.1 Representation of first-order theories 54
4.4.2 Partial evaluation of propositional formulas 58
4.4.3 Hardware acceleration 61
4.4.4 Accelerated primitives 64
4.4.5 Example 69
4.5 Comparison against similar software 71

5 Applications 73

5.1 Lattices 73
5.1.1 Removing variables 73
5.1.2 Definable partitions 76
5.1.3 Irreducible components 79
5.1.4 Lattice counting: program implementation 82
5.2 Tournament graphs 83
5.2.1 Graphs and tournament 83
5.2.2 Kauer's graph 84
5.2.3 Random graphs with n vertices 85
5.2.4 Random tournaments 85
5.2.5 Example 1: Erdös, [1963] 85
5.2.6 Example 2: Kauer's tournament 86
5.2.7 Supplement A: 0–1 Law 87
5.2.8 Supplement B: Further problems 89

6 Conclusion 90

6.1 Summary of results 90
6.2 Related work 92
6.3 Future work 93

References 94

Appendix A Source code listings 97

List of figures

2.1 Terms of a first-order language L . . . 7
2.2 Proof theoretical hierarchy of first-order formulas . . . 13
2.3 Homomorphism theorem for terms . . . 20

4.1 AST for $p \wedge q \vee \neg r$ . . . 55

List of tables

4.1 Bitwise operations over ASTs 58
4.2 A partitioning of free Boolean vectors of $\mathbf{\Omega}_5$ 64

List of listings

3.1 Permutations$(n, 3)$ 41
4.1 Partial evaluation visitor 59
4.2 Evaluation function template 63
4.3 Template of the `eval` kernel 67
4.4 Template of the `count` kernel 68
4.5 Template of the `findmodel` kernel 69
4.6 Latin square properties 70
4.7 Sudoku property 70

Nomenclature

Acronyms / Abbreviations

AST abstract syntax tree

CNF conjunctive normal form

DNF disjunctive normal form

DSL domain-specific language

GPU graphics processing unit

PR1 first-order predicate calculus

Chapter 1

Introduction

The theme of the dissertation is related to certain theoretical aspects of computer science and practical and effective use of computers. More precisely, the theme is in the field of Finite Model Theory (FMT) and its main contribution is a development of a parallel software system for solving problems of generating and counting finite structures of first order theories.

FMT is a subfield of model theory, one of the key disciplines of modern mathematical logic. In contrast to the classical model theory, FMT besides contributions to the theory of finite structures (e.g., well-known 0–1 theorem) has become a very important instrument in computing and applications. Examples of applications include solving various problems of finite combinatorics, artificial intelligence, databases, formal languages and analysis of logic circuits. Development of appropriate software platforms is particularly important in these activities since they allow us to represent and solve problems of finite model theory. In most cases this type of software uses the formalism of first-order predicate calculus (PR1). Research in the dissertation exactly refers to the development of such a system, which is largely based on parallel programs that are designed to run on graphics cards with a large number of processor cores.

Theoretical background of this research lies in the following fields:

Model theory with emphasis on finite model theory. A short introduction to this subject will be given: first order languages and theories, models, satisfaction relation, and method of new constants. In addition, some proofs will be outlined.

Algebraic and combinatorial constructions It will appear that the use of some combinatorial constructs will have fundamental role in this research. Therefore, in this context we shall introduce and discuss group actions on models and the so called inversion formulas.

Free algebras and equational logic Implementation of our software system is heavily based on free Boolean algebras and their free generators. Therefore, we give a concise

introduction to free algebras, with emphasis on examples and constructions of free Boolean algebras. As an illustration of this method we gave a novel proof of a Birkhoff HSP theorem and Completeness theorem for Equational logic.

The main goal of the thesis is to develop a programming system based on the above principles and to give some applications of it.

Programming systems As a part of the thesis we developed a software system for finding finite models of theories in first-order predicate logic and some of its extensions. It is implemented as a parallel software that can run both on CPUs as well as GPUs. We gave a comparative analysis of our software to other similar softwares, particularly to Mace4. The framing language of our system is Python, while its parallel core is implemented in OpenCL. The main idea was to implement an abstract processor with 2^n bits. The theoretical background of this abstract processor are free Boolean vectors. Besides that, we also implemented:

- Translation of formulas of predicate logic to formulas of proposition calculus.
- Reduction of very large propositional expressions limited only by hardware resources.
- The programming system that is embedded in Python environment. In other words, commands of the system can by used in Python programs as any other native construct.

Applications We have two main examples of applications. The first one concerns partial orders and in particular lattices. We were especially interested in the so called counting problem—finding the number of labeled and unlabeled structures. The second example deals with the random graphs, an area mostly developed by Erdös and Rénei. Of the particular importance in the study of these graphs is the so called 0–1 Low (Glebski, Kogan, Liagonki, Talanov[1969], rediscovered by Fagin[1976]). We proved certain properties of special tournaments (oriented complete graph). For example, we proved the existence of a Kauer tournament for $n \geqslant 7$.

Chapter 2

Model theory

This chapter is designed as an overview of the main concepts and definitions of classical model theory. Of course, we could not cover all the topics in model theory, but the most important constructions and theorems of model theory and their proofs are outlined. To large extent, we relied on several monographs in this field, such as [7] and [29].

Boolean algebras will play an important role in this dissertation. The use of Boolean algebras in model theory is prolific. They are applied in many model-theoretic constructions, but we have also used them here in the design of our software system.

Basic constructions of models are presented such as the method of constants, elementary equivalence of models and types. A few words are devoted to extensions of first-order logic.

We presuppose some parts of the naive set theory. This includes the basic properties of ordinal and cardinal numbers and partially, their arithmetic. The cardinal number of a set X is denoted by $|X|$, and the set of all subsets of X by $P(X)$.

We have adopted Von Neuman representation of ordinals, so we have taken that every ordinal is the set of all the smaller ordinals. Therefore

$$0 = \emptyset,\ 1 = \{0\},\ 2 = \{0, 1\},\ \ldots,\ \omega_0 = \{0, 1, 2, \ldots\},\ \ldots.$$

Here $\emptyset$ denotes the empty set. The set of all natural numbers is denoted by ω_0, i.e. $\omega = \omega_0 = \{0, 1, 2, \ldots\}$. We do not distinguish ordinal numbers ω_α and cardinal numbers $\aleph_\alpha$.

If $f: A \to B$ is a mapping from a set A into a set B and $X \subseteq A$, then

- $f|X$ denotes the restriction of f to the set X,
- $f[X] = \{f(x): x \in X\}$, but sometimes we write $f(X)$ for $f[X]$ as well,
- $f\mathbf{x}$ or $f(\mathbf{x})$ stands for the sequence $fx_1, fx_2, \ldots, fx_n$, where $\mathbf{x}$ denotes the sequence $x_1, x_2, \ldots, x_n$.

Our metatheory is based on the ZFC set theory, and we shall not point out explicitly when we use, for example, the Axiom of Choice or its equivalents. However, all exceptions will be indicated, as the use of the Continuum Hypothesis or some weaker variants of the Axiom of Choice.

Final remarks are on usage and signs. The word “iff” is often used instead of the phrase “if and only if”. The end of a proof is indicated by □.

2.1 Introduction to model theory

Model theory is often defined as a formal logic and universal algebra. More detailed analysis shows that model theory is the study of the relationship between syntactical objects on the one hand and the structures of a set-theoretical nature on the other hand, or in other words, between formal languages and their interpretations.

Therefore, two areas of logic, syntax and semantics, both have a role to play in this subject. While syntax is concerned mainly with the formation rules of formulas, sentences and other syntactical objects, semantics bears on the meaning of these notions. One of the most important concepts of model theory is the satisfaction relation, denoted by $\models$, a relation between mathematical structures and sentences.

Model theory was recognized as a separate subject during the thirties of the XX century in the works of Thoralf Skolem (1887-1963, Norwegian), Alfred Tarski (1901-1983, Polish), Kurt Gödel (1906–1978, Austrian), Anatoly Malcev (1909–1967, Russian) and their followers. Since then, this field has developed vigorously, and was applied in many other branches of mathematics: algebra, set-theory, nonstandard analysis, computer science and even mathematical economy.

We can speak of model theory of any kind of logic, but we shall study model theory of first-order predicate calculus.

2.2 First-order languages

A first order language is any set L of constant symbols, function symbols and relation symbols. Each of the relation and function symbols has some definite, finite number of argument places. Sometimes it is convenient to consider constant symbols as function symbols with zero argument places.

According to our classification, we have

$$L = \mathrm{Fnc}_L \cup \mathrm{Rel}_L \cup \mathrm{Const}_L,$$

where:

- $\mathrm{Fnc}_L = \{s \in L\colon s \text{ is a function symbol of } L\}$,
- $\mathrm{Rel}_L = \{s \in L\colon s \text{ is a relation symbol of } L\}$,
- $\mathrm{Const}_L = \{s \in L\colon s \text{ is a constant symbol of } L\}$.

All these three sets are pairwise disjoint, and each of them may be an empty set. Namely, we shall deal only with logic with equality.

If L and L' are first order languages, and $L \subseteq L'$, then L' is called an expansion of the language L, while L is called a reduct of L'. If $L' \setminus L$ is a set of constant symbols, then we say that L' is a simple expansion of L.

The arity–function $\mathrm{ar}\colon L \to \omega$ assigns to each $s \in L$ its length, i.e., the number of argument places. By the remark above, if $s \in Const_L$, we define $\mathrm{ar}(s) = 0$, while for $s \in \mathrm{Fnc}_L \cup \mathrm{Rel}_L$, we have $\mathrm{ar}(s) \geqslant 1$.

In most cases it will be clear from the context what the lengths of the symbols of L are, so in such cases the arity function will not be mentioned explicitly. However, we take $\mathrm{Fnc}_L^k = \{F \in \mathrm{Fnc}_L\colon \mathrm{ar}(F) = k\}$. A similar meaning has Rel_L^k for relation symbols of L.

Example 2.2.1. The language of ordered fields is $L = \{+, -, \cdot, \leqslant, 0, 1\}$. Here $\mathrm{Fnc}_L = \{+, -, \cdot\}$, $\mathrm{Rel}_L = \{\leqslant\}$, $\mathrm{Const}_L = \{0, 1\}$.

2.3 Terms and formulas

Terms and formulas of a first-order language L are special finite sequences of the symbols of L and the logical symbols of the first-order predicate calculus (which shall be abbreviated PR1).

Logical symbols of PR1 are logical connectives and quantifiers:

- $\wedge$ – and, $\vee$ – or, $\Rightarrow$ – implication, $\Leftrightarrow$ – equivalence,
- $\neg$ – negation and the equality sign $\equiv$,
- $\forall$ – universal quantifier and $\exists$ – existential quantifier.

Finally, we have an infinite sequence of variables $v_1, v_2, \ldots$.

The unique readability of terms and formulas must be provided, so some auxiliary symbols are used, the left and right parenthesis and the comma sign: **( ) ,**.

Metavariables are $x, y, z, x_0, y_0, z_0, \ldots$, and they may denote any variable v_i, $i \in \omega$, i.e. the domain of metavariables is the set $\mathrm{Var} = \{v_1, v_2, \ldots\}$. Metaequality is another important such sign and it will be denoted by $=$.

Terms, or algebraic expressions of a language L can be described inductively:

- Variables and constant symbols of L are terms.
- If $F \in \mathrm{Fnc}_L$ is of the length n, and $t_1, \ldots, t_n$ are terms of L, then $F(t_1, \ldots, t_n)$ is a term of L.
- Every term of L is obtained by a finite number of applications of the previous two rules.

A somewhat more formal (recursive) definition of this notion is:

Definition 2.3.1. A term of a language L is any element t of Term_L, where Term_L is defined in the following way:

$$\begin{aligned} T_0 &= \mathrm{Var} \cup \mathrm{Const}_L, \\ T_{n+1} &= T_n \cup \{F(t_1, \ldots, t_k) \colon t_1, \ldots, t_k \in T_n, F \in \mathrm{Fnc}_L^k, k \in \omega\}, \\ \mathrm{Term}_L &= \bigcup_{n \in \omega} T_n. \end{aligned}$$

Standard rules are applied on terms: rules about deleting parenthesis, special notation for binary function symbols, possible priority of function symbols, etc.

The complexity function $\mathrm{co} \colon \mathrm{Term}_L \to \omega$ is a measure of the complexity of terms. It is defined in the following way:

$$\mathrm{co}(t) = \begin{cases} 0 & \text{if } t \in T_0, \\ n,\ n \in \omega & \text{if } t \in T_n \setminus T_{n-1}. \end{cases}$$

The complexity of terms, for example, of the language $L = \{F, G\}$, where F and G are the binary function symbols, can be visualized as in Fig. 2.1.

Formulas of the first-order language L are defined in a similar manner. First, the atomic formulas are defined:

Definition 2.3.2. A string φ is an atomic formula of a language L, if and only if φ has one of the following forms:

(i) $u \equiv v$, u, v are terms of L,

(ii) $R(t_1, \ldots, t_n)$, $R \in \mathrm{Rel}_L^n$ and $t_1, \ldots, t_n$ are terms of L.

Let At_L denote the set of the atomic formulas of L. Then by the previous definition we have

$$\begin{aligned} \mathrm{At}_L = \{u \equiv v \colon u, v \in \mathrm{Term}_L\} \cup \\ \{R(t_1, \ldots, t_n) \colon n \in \omega, R \in \mathrm{Rel}_L^n, t_1, \ldots, t_n \in \mathrm{Term}_L\}. \end{aligned}$$

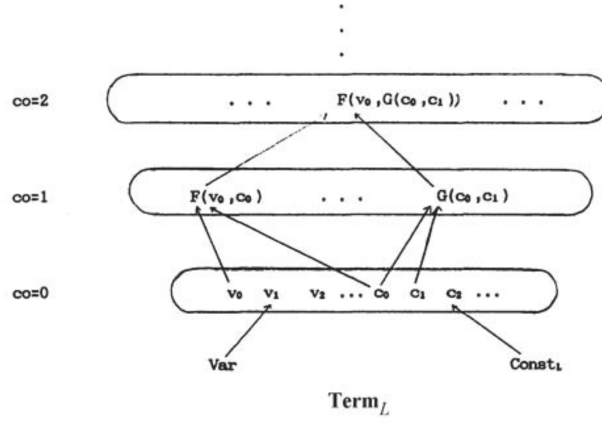


Fig. 2.1 Terms of a first-order language L

Formulas of a language L are also defined inductively by the use of an auxiliary sequence $\mathcal{F}_n$, $n \in \omega$, of sets of strings of L:

$$
\begin{aligned}
\mathcal{F}_0 &= \mathrm{At}_L, \\
\mathcal{F}_{n+1} &= \mathcal{F}_n \cup \{(\varphi \wedge \psi) \colon \varphi, \psi \in \mathcal{F}_n\} \\
&\quad \cup \{(\varphi \vee \psi) \colon \varphi, \psi \in \mathcal{F}_n\} \\
&\quad \cup \{(\neg\varphi) \colon \varphi \in \mathcal{F}_n\} \\
&\quad \cup \{(\varphi \Rightarrow \psi) \colon \varphi, \psi \in \mathcal{F}_n\} \\
&\quad \cup \{(\varphi \Leftrightarrow \psi) \colon \varphi, \psi \in \mathcal{F}_n\} \\
&\quad \cup \{(\forall x\varphi) \colon x \in \mathrm{Var}, \varphi \in \mathcal{F}_n\} \\
&\quad \cup \{(\exists x\varphi) \colon x \in \mathrm{Var}, \varphi \in \mathcal{F}_n\}, \; n \in \omega, \\
\mathrm{For}_L &= \bigcup_{n \in \omega} \mathcal{F}_n.
\end{aligned}
$$

Definition 2.3.3. Then the elements of the set For_L are defined as formulas of the language L.

It is not difficult to see that the formulas satisfy the following conditions:

- Atomic formulas are formulas.
- If φ and ψ are formulas of L, and x is a variable, then

  $$(\varphi \wedge \psi),\ (\varphi \vee \psi),\ (\neg\varphi),\ (\varphi \Rightarrow \psi),\ (\varphi \Leftrightarrow \psi),\ (\exists x\varphi),\ (\forall x\varphi)$$

  are also formulas of L.

- Every formula of L is obtained by a finite number of use of the previous two rules.

In order to measure the complexity of formulas, we shall extend the complexity function co to formulas as well. Therefore, $\mathrm{co}\colon \mathrm{For}_L \to \omega$ is defined inductively in the following way:

$$\mathrm{co}(\varphi) = \begin{cases} 0 & \text{if } \varphi \in \mathrm{At}_L, \\ n & \text{if } \varphi \in \mathrm{For}_n \setminus \mathrm{For}_{n-1},\ n \in \omega \setminus \{0\}. \end{cases}$$

As in the case of terms, we suppose that the reader is familiar with the basic conventions about formulas: the use of rules on deleting parenthesis, priority of logical connectives, etc. In addition, we shall shrink blocks of quantifiers, for example instead of $\forall x_0 \forall x_1 \dots \forall x_n \varphi$ we shall write $\forall x_0 x_1 \dots x_n \varphi$ whenever appropriate.

The notion of a free occurrence of variables allows us to describe precisely the variables of a formula φ which are not in the scope of the quantifiers.

Definition 2.3.4. The set $\mathrm{Fv}(\varphi)$ of variables which have free occurrences in a formula φ of L is introduced inductively by the complexity of φ in the following way:

- If $\varphi \in \mathrm{At}_L$, then $\mathrm{Fv}(\varphi)$ is the set of variables which occur in φ.
- $\mathrm{Fv}(\neg\varphi) = \mathrm{Fv}(\varphi)$.
- $\mathrm{Fv}(\varphi \wedge \psi) = \mathrm{Fv}(\varphi \vee \psi) = \mathrm{Fv}(\varphi \Rightarrow \psi) = \mathrm{Fv}(\varphi \Leftrightarrow \psi) = \mathrm{Fv}(\varphi) \cup \mathrm{Fv}(\psi)$.
- $\mathrm{Fv}(\exists x\varphi) = \mathrm{Fv}(\forall x\varphi) = \mathrm{Fv}(\varphi) \setminus \{x\}$.

The elements of the set $\mathrm{Fv}(\varphi)$ are called free variables of the formula φ, while the other variables which occur in φ are called bounded. If$\varphi \in \mathrm{For}_L$, then the notation $\varphi(x_0, \dots, x_n)$, or $\varphi x_0 \dots x_n$ is used to denote that free variables of φ are among the variables $x_0, \dots, x_n$.

Example 2.3.1. If $\varphi = (\neg x \equiv 0 \Rightarrow \exists y(x \cdot y \equiv 1))$ then $\mathrm{Fv}(\varphi) = \{x\}$, so x is a free variable of φ and y is a bounded variable of φ.

Formulas φ which do not contain free variables, i.e. $\mathrm{Fv}(\varphi) = \emptyset$, are called sentences. The formulas

$$0 \equiv 1 \quad \text{and} \quad \forall x(\neg x \equiv 0 \Rightarrow \exists y(x \cdot y \equiv 1))$$

are examples of sentences of the language $L = \{\cdot, 0, 1\}$, where $\cdot$ is a binary function symbol.

The set of all sentences of L is denoted by Sent_L. The cardinal number of For_L is denoted by $\|L\|$, therefore $\|L\| = |For_L|$. It is not difficult to see that for every first-order language L we have

$$\|L\| = \max\{|L|, \aleph_0\}.$$

2.4 First-order theories

The definition of the notion of a first-order theory is simple:

Definition 2.4.1. A theory of a first order language L is any set of sentences of L.

Therefore, a set T is a theory of L iff $T \subseteq \text{Sent}_L$. In this case elements of T are called axioms of T.

The main notion connected to the concept of a theory is the notion of a proof in the first-order logic. There are several approaches to the formalization of the notion of a proof. For example, Gentzen's systems are very useful for the analysis of the proof-theoretical strength of mathematical theories. The emphasis in Gentzen's approach is on deduction rules, as distinct from Hilbert-oriented systems, where the stress is on the axioms.

Hilbert style formal systems are more convenient in model theory, so we shall confine our attention to them. They consist of logic axioms, inference rules and special axioms. Now we shall see how they look like for a first-order language L.

The logic axioms are:

Sentential axioms. These axioms are derived from propositional tautologies by the simultaneous substitution of propositional letters by formulas of L.

Identity axioms. If $\varphi \in \text{For}_L$, $t \in \text{Term}_L$, $x \in \text{Var}$, then $\varphi(t/x)$ denotes the formula obtained from φ by substituting the term t for each free occurrence of x in φ. Sometimes, we shall use the abridged form $\varphi(t)$ or φt, instead of $\varphi(t/x)$. Now we shall list the identity axioms:

$$x \equiv x,$$

and for $n \in \omega$, we also have the following scheme axioms

$$x_1 \equiv y_1 \wedge x_2 \equiv y_2 \wedge \ldots \wedge x_n \equiv y_n \Rightarrow tx_1x_2\ldots x_n \equiv ty_1y_2\ldots y_n,$$
$$x_1 \equiv y_1 \wedge x_2 \equiv y_2 \wedge \ldots \wedge x_n \equiv y_n \Rightarrow \varphi x_1x_2\ldots x_n \Leftrightarrow \varphi y_1y_2\ldots y_n.$$

Quantifier axioms. If $\varphi \in \text{For}_L$, $t \in \text{Term}_L$, $x \in \text{Var}$ then

$$\forall x\varphi x \Rightarrow \varphi t, \quad \varphi t \Rightarrow \exists x\varphi x,$$

where φt is obtained from φx by freely substituting each free occurrence of x in φx by the term t.

In addition, let φ and ψ be formulas of L, then the rules of inference are:

Modus Ponens:

$$\frac{\varphi, \quad \varphi \Rightarrow \psi}{\psi}.$$

Generalization rules. Provided x does not occur free in φ:

$$\frac{\varphi \Rightarrow \psi}{\varphi \Rightarrow \forall x \psi},$$

$$\frac{\psi \Rightarrow \varphi}{\exists x \psi \Rightarrow \varphi}.$$

And finally, the special axioms are axioms of a theory T.

A proof of a sentence φ in a theory T of a language L is every sequence $\psi_1, \psi_2, \ldots, \psi_n$ of formulas of the language L such that $\varphi = \psi_n$ and each formula ψ_i, $i = 1, \ldots, n$, is a logical axiom, or an axiom of T, or it is derived by inference rules from the preceding members of the sequence.

If there exists a proof of φ in T, then φ is called a theorem of T, and in this case we use the notation $T \vdash \varphi$. The relation $\vdash$ between theories and formulas of a language L is the provability relation. If $T = \emptyset$, then we simply write $\vdash \varphi$ instead of $\emptyset \vdash \varphi$, and φ is called a theorem of the first-order predicate calculus. If φ is not a theorem of T, then we shall write $\sim T \vdash \varphi$ or $T \nvdash \varphi$ for short.

Formulas of the form $\varphi \wedge \neg\varphi$ are called contradictions. A theory T is consistent if there is no contradiction ψ such that $T \vdash \psi$.

Another important property which theories may have is completeness. A theory T of a language L is complete if for each sentence φ of L either $T \vdash \varphi$ or $T \vdash \neg\varphi$.

Finally, T is deductively closed if T contains all of its theorems.

There is a group of first-order notions which are related to the effective computability. We shall suppose that the reader has some basic ideas of the effective computability and arithmetical coding. So, for this purpose, if $\varphi \in \mathrm{For}_L$, then $\ulcorner\varphi\urcorner$ denotes the code of formula φ. We remind that the code $\ulcorner\varphi\urcorner$ is a unique positive integer assigned to φ. A similar notation is applied to other syntactical objects (terms, elements of L, etc.).

A first order language L is recursive, if the set $\ulcorner L\urcorner = \{\ulcorner s\urcorner : s \in L\}$ is recursive. Similarly, L is recursively enumerable if $\ulcorner L\urcorner$ is a recursively enumerable set.

A theory T of the language L is finitely axiomatizable, if T has a finite set of axioms. A generalization of this notion is the concept of an axiomatic theory.

Definition 2.4.2. A theory T is axiomatic or recursive if T i.e. $\{\ulcorner\varphi\urcorner : \varphi \in T\}$ is a recursive set of sentences.

The definitions of notions introduced in this way can be broadened. Namely, two theories T and S of the same language L are equivalent, if they have the same theorems. Then a theory T is considered to be also finitely axiomatizable (axiomatic), if there is a theory S equivalent to T which has a finite set of axioms.

A first-order theory T is decidable, if the set of all theorems of T is decidable (i.e. recursive) set, otherwise T is undecidable. The most interesting mathematical theories are undecidable. However, the following proposition gives a test of decidability for certain theories.

Theorem 2.4.1. *Suppose T is an axiomatic and complete theory of a recursive language L. Then T is decidable.*

Proof. Let T' be the set of all the theorems of T. Since T is complete, for each $\varphi \in \mathrm{Sent}_L$ we have $\varphi \in T'$ or $\neg\varphi \in T'$. If for some sentence φ it holds that $\varphi, \neg\varphi \in T'$, then $T' = \mathrm{Sent}_L$, and since Sent_L is a recursive set, it follows T' is recursive as well.

Suppose the second, more interesting case holds, i.e., that T is a consistent theory. Since T is recursive, the set (of all the codes) of proofs may be effectively listed. By the completeness of T, for each sentence φ of L either φ or $\neg\varphi$ should occur as the last member of a proof in the list. In the first case, φ is a theorem of T, while in the second case, φ is not a theorem of T by the consistency of T.

The just described property of T defines an algorithm for decidability of $T \vdash \varphi$, where $\varphi \in \mathrm{Sent}_L$:

> Generate all the proofs of T, and look at the end of each proof until one of the formulas φ, $\neg\varphi$ appears. If φ occurs then $T \vdash \varphi$, otherwise $T \vdash \neg\varphi$.

This search will stop since either $T \vdash \varphi$, or $T \vdash \neg\varphi$. □

Here are several elementary, but important, theorems from logic without proofs.

Theorem 2.4.2 (Deduction Theorem)**.** *Suppose T is a theory of a language L and $T \vdash \varphi$ where $\varphi \in \mathrm{For}_L$. Then, there are sentences $\theta_0, \theta_1, \ldots, \theta_n \in T$ such that*

$$\vdash \theta_0 \wedge \theta_1 \wedge \ldots \wedge \theta_n \Rightarrow \varphi.$$

As a consequence of this theorem we have that a first order theory T is consistent iff every finite subset of T is consistent.

Lemma 2.4.1 (Lemma on the new constant)**.** *Let T be a theory of a language L, and assume c is a constant symbol not occurring in L. Then for every formula $\varphi(x)$ of L we have*:

$$\text{if } T \vdash \varphi(c), \text{ then } T \vdash \forall x \varphi(x).$$

Proof. The proof of this lemma is very easy. If in the proof of $\varphi(c)$ from T, the constant symbol c is replaced by a variable y, which does not occur in that proof, then we shall obtain a proof of $\varphi(y)$ from T. By the inference rule of generalization, the lemma then follows at once. □

A formula φ of a first-order language L is in a prenex normal form (PNF), if φ is of the form $Q_1 y_1 Q_2 y_2 \ldots Q_n y_n \psi$, where ψ is a formula without quantifiers, and $Q_1, Q_2, \ldots, Q_n$ are some of the quantifiers $\forall$, $\exists$. In this case the formula ψ is called a matrix.

Theorem 2.4.3 (PNF Theorem)**.** *For every formula φ of a first order language L, there exists a formula ψ of L in a prenex normal form, such that $T \vdash \varphi \Leftrightarrow \psi$.*

Another important notion is related to this theorem. This is the so-called proof-theoretical hierarchy of formulas of a language L.

Definition 2.4.3. Let L be a first order language. Then:

$$\begin{aligned} \textstyle\sum_0^0 = \prod_0^0 &= \{\varphi \in \mathrm{For}_L \colon \varphi \text{ does not contain quantifiers}\}, \\ \textstyle\sum_{n+1}^0 &= \left\{\exists x_1 \ldots x_k \varphi \colon k \in \omega, \varphi \in \textstyle\prod_n^0\right\}, \\ \textstyle\prod_{n+1}^0 &= \left\{\forall x_1 \ldots x_k \varphi \colon k \in \omega, \varphi \in \textstyle\sum_n^0\right\}. \end{aligned}$$

If $\varphi \in \sum_n^0$ then φ is called a $\sum_n^0$-formula, and if $\varphi \in \prod_n^0$, then φ is a $\prod_n^0$-formula. If φ is a $\sum_1^0$-formula, then φ is also called an existential formula, while if φ is a $\prod_1^0$-formula, then φ is called a universal formula.

The sequences $\sum_n^0$ and $\prod_n^0$ of formulas of L define a proof-theoretical hierarchy of formulas of L. By PNF Theorem every formula φ of L is equivalent to a formula ψ such that either $\psi \in \sum_n^0$ or $\psi \in \prod_n^0$. Then φ is called a $\sum_n^0$-formula and respectively $\prod_n^0$-formula. If φ is a formula of L and for some $n \in \omega$ there is a $\psi \in \sum_n^0$ and a $\theta \in \prod_n^0$ both equivalent to φ then φ is called a Δ_n^0-formula.

The main properties of the proof-theoretical hierarchy are described in the Fig. 2.2.

2.5 Examples of theories

We shall give several examples of first-order theories. Most examples are from working mathematics, and we shall consider some cases in greater detail. For every example, we shall exhibit explicitly the corresponding language L in which the axioms of the theory are written down.

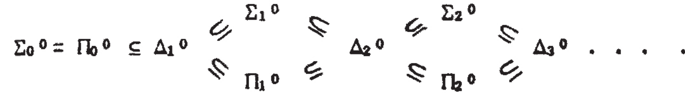


Fig. 2.2 Proof theoretical hierarchy of first-order formulas

Example 2.5.1. Pure predicate calculus with identity, J_0. For this theory we have:

$$L = \emptyset,\ T = \emptyset.$$

Theorems of J_0 are exactly the theorems of PR1 which contain only logical symbols. Here are several interesting examples of sentences which can be written in L:

$$\begin{aligned}
\sigma_1 &= \exists x_1 \forall x (x \equiv x_1),\\
\sigma_2 &= \exists x_1 x_2 (\neg x_1 \equiv x_2 \wedge \forall x (x \equiv x_1 \vee x \equiv x_2)),\\
&\ \vdots\\
\sigma_n &= \exists x_1 \ldots x_n (\bigwedge_{1 \leqslant i < j \leqslant n} (\neg x_i \equiv x_j) \wedge \forall x (\bigvee_{1 \leqslant i \leqslant n} (x \equiv x_i))).\\
\tau_1 &= \exists x_1 (x_1 \equiv x_1),\\
\tau_2 &= \exists x_1 x_2 (\neg x_1 \equiv x_2),\\
&\ \vdots\\
\tau_n &= \exists x_1 \ldots x_n (\bigwedge_{i<j} (\neg x_i \equiv x_j)).
\end{aligned}$$

σ_n – “There are exactly n elements”.

τ_n – “There are at least n elements”.

Example 2.5.2. The theory of linear ordering, LO. In this case we have: $L_{\mathrm{LO}} = \{\leqslant\}$, $\leqslant$ is a binary relation symbol. Axioms of T are:

$LO.1$	$x \leqslant x$,	reflexivity,
$LO.2$	$x \leqslant y \wedge y \leqslant z \Rightarrow x \leqslant z$,	transitivity,
$LO.3$	$x \leqslant y \wedge y \leqslant x \Rightarrow x \equiv y$,	antisymmetry,
$LO.4$	$x \leqslant y \vee y \leqslant x$,	linearity.

A theory PO whose axioms are LO.1–3 is called a theory of partial ordering. The binary relation symbol $<$ is introduced by the definition axiom: $x < y \Leftrightarrow x \leqslant y \wedge x \not\equiv y$.

Example 2.5.3. The theory of dense linear ordering without endpoints, DLO. The language of this theory is the same as in the case of LO, while the axioms DLO are the axioms of LO plus the following sentences:

$$\exists x \exists y\ x \not\equiv y, \quad \forall x \exists y\ x < y, \quad \forall x \exists y\ y < x,$$
$$\forall x \forall y \exists z (x < y \Rightarrow x < z \wedge z < y).$$

It is not difficult to see that for each $n \in \omega \setminus \{0\}$, DLO $\vdash \tau_n$, where τ_n is the sentence from Example 2.5.1.

Example 2.5.4. The theory of Boolean algebras, BA. The language of this theory is $L_{\mathrm{BA}} = \{+, \cdot, ', \leqslant, 0, 1\}$, where $+$ and $\cdot$ are binary function symbols, $'$ is a unary function symbol, $\leqslant$ is a binary relation symbol, while 0, 1 are constant symbols. The axioms of BA are:

1. $(x + y) + z \equiv x + (y + z)$,
2. $x + y \equiv y + x$,
3. $x + 0 \equiv x$,
4. $x + x' \equiv 1$,
5. $0 \not\equiv 1$,
6. $x \leqslant y \Leftrightarrow x \equiv x \cdot y$.

1′. $(x \cdot y) \cdot z \equiv x \cdot (y \cdot z)$,
2′. $x \cdot y \equiv y \cdot x$,
3′. $x \cdot 1 \equiv x$,
4′. $x \cdot x' \equiv 0$,

The following notation is also used for Boolean operations. Namely, the symbols $\vee$ and $\wedge$ are often used for $+$ and $\cdot$ respectively. The sign $'$ is used unchanged, but $\bar{x}$ can be used instead as well. For example, the term $(x' \cdot y) + z$ in the new notation may be written as $(\bar{x} \wedge y) \vee z$.

It is easy to infer in BA the axioms of partial order in respect to the relation symbol $\leqslant$. We have the following important theorems of the theory BA:

Theorem 2.5.1. *Let* sup *and* inf *denote the order supremum and infimum in respect to* $\leqslant$*. Then the next identities are theorems of* BA:

$$\sup\{x_1, x_2, \dots, x_n\} \equiv \sum_{i \leqslant n} x_i, \quad \inf\{x_1, x_2, \dots, x_n\} \equiv \prod_{i \leqslant n} x_i.$$

Theorem 2.5.2. *For each* $t \in \mathrm{Term}_{L_{\mathrm{BA}}}$,

$$\mathrm{BA} \vdash t(x_1, x_2, \dots, x_n) \equiv \sum_{\alpha \in 2^n} t(\alpha_1, \alpha_2, \dots, \alpha_n) x_1^{\alpha_1} \dots x_n^{\alpha_n}$$

where $2^n = \{\alpha \mid \alpha : n \to 2\}$, *and* $x^0 = x'$, $x^1 = x$.

This property of Boolean terms is proved by induction on the complexity of terms.

All the examples we have listed are axiomatic theories, i.e., with recursive sets of axioms. Also, all except the last example, are finitely axiomatizable theories.

Theories J_0, LO, DLO, and BA are decidable. An example of a theory that is not decidable would be the Peano arithmetic, PA. This is certainly the most famous example of an undecidable theory.

Interestingly, the Presburger arithmetic, a subtheory of PA, is also a decidable and complete theory. The first example of a program not dealing with numbers but only with symbols was the implementation of the decision procedure for Presburger arithmetic (Martin Davis, 1954).

2.6 Models

We have dealt in the previous sections mainly with syntactical notions. On the other hand, the most important concept in model theory is the idea of an operational-relational structure, or simply a model of a first-order language L.

Customary mathematical structures such as groups, fields, ordered fields, and the structure of natural numbers are examples of models. When studying the properties of models, a distinctively important role is played by the concept of formal language used to make precise the set of symbols and rules used to build formulas and sentences.

The main reason for introducing formulas is to describe properties of models. Therefore, it is not astonishing that some properties of models are often consequences of the structure of sentences or classes of sentences. The proofs of such features of models are often called model-theoretical proofs.

Using the methods of model theory many open mathematical problems have been solved. One such famous problem is the consistent foundation of Leibnitz Analysis, a problem which stood open for 300 years. Abraham Robinson gave a simple but ingenious solution, and thanks to him there is now a whole new methodology which is equally well applied to topology, algebra, probability theory, and practically to all mathematical fields where infinite objects appear.

Definition 2.6.1. A model is every structure $\mathbf{A} = (A, \mathcal{R}, \mathcal{F}, C)$ where A is a nonempty set (the domain of $\mathbf{A}$), $\mathcal{R}$ is a set of relations over A, $\mathcal{F}$ is a family of operations over A, and C is a set of constants of A.

By this definition of a model we have:

If $R \in \mathcal{R}$, then there is $n \in \omega$, such that $R \subseteq A^n$, i.e., R is a relation over A of a length n. The length of R is denoted by $\mathrm{ar}(R)$.

If $F \in \mathcal{F}$, then there is an $n \in \omega$ such that $F\colon A^n \to A$, i.e., F is an n-ary operation over A. The length of F is denoted by $\mathrm{ar}(F)$.

Finally, $C \subseteq A$.

If $\mathcal{R}$, $\mathcal{F}$, and C are finite sets, for example

$$\mathcal{R} = \{R_1, R_2, \ldots, R_m\}, \quad \mathcal{F} = \{F_1, F_2, \ldots, F_n\}, \quad C = \{a_1, a_2, \ldots, a_k\},$$

then $\mathbf{A}$ may be denoted as

$$\mathbf{A} = (A, F_1, F_2, \ldots, F_n, R_1, R_2, \ldots, R_m, a_1, a_2, \ldots, a_k).$$

If these sets are indexed, i.e., $\mathcal{R} = \langle R_j : j \in J\rangle$, $\mathcal{F} = \langle F_i : i \in I\rangle$, $C = \langle a_k : k \in K\rangle$, we can also use the notation: $\mathbf{A} = (A, F_i, R_j, a_k)_{i \in I, j \in J, k \in K}$.

Example 2.6.1. Some example models:

1. The ordered field of real numbers: $\mathbf{R} = (R, +, \cdot, -, \leqslant, 0, 1)$. Here, $\mathcal{F} = \{+, \cdot, -\}$, $\mathcal{R} = \{\leqslant\}$, $\mathrm{ar}(\leqslant) = 2$, $\mathrm{ar}(+) = ar(\cdot) = 2$, $\mathrm{ar}(-) = 1$ and $C = \{0, 1\}$.
2. The structure of natural numbers: $\mathbf{N} = (N, +, \cdot, ', \leqslant, 0)$.
3. The field of all subsets of a set X: $\mathbf{P}(X) = (P(X), \cup, \cap, {}^c, \subseteq, X)$, where $P(X) = \{Y : Y \subseteq X\}$, and for $Y \in P(X)$, $Y^c = X \setminus Y$.

Models are interpretations of first-order languages. To see that, let L be a first-order language and A a non-empty set. An interpretation of L into the domain A is every mapping I with the domain L, and values determined as follows:

If $R \in \mathrm{Rel}_L$, then $I(R)$ is a relation of A of a length $\mathrm{ar}(R)$.

If $F \in \mathrm{Fnc}_L$, then $I(F)$ is an operation of A of a length $\mathrm{ar}(F)$.

If $c \in \mathrm{Const}_L$ then $I(c) \in A$.

Therefore, every interpretation I of a language L into a domain A determines a unique model $\mathbf{A} = (A, I(\mathrm{Rel}_L), I(\mathrm{Fnc}_L), I(\mathrm{Const}_L))$. So introduced model is written simply as

$$\mathbf{A} = (A, I), \quad \text{or} \quad \mathbf{A} = (A, s^{\mathbf{A}})_{s \in L},$$

where for $s \in L$, $s^{\mathbf{A}} = I(s)$.

We see in Example 2.6.1 that $\mathbf{R}$ is a model of the language of ordered fields, while $\mathbf{N}$ is a model of the language of Peano arithmetic and finally $\mathbf{P}(X)$ is a model of the language of the theory of Boolean algebras.

From now on letters $\mathbf{A}, \mathbf{B}, \mathbf{C}, \dots$ will be reserved for models and letters $A, B, C, \dots$ for their domains.

If L is a language and $\mathbf{A}$ is a model of L, then $s \in L$ and $s^{\mathbf{A}}$ denote objects of a different nature. However, if the context allows, we shall use the same sign to denote a symbol of L and its interpretation in $\mathbf{A}$. Therefor the superscript $^{\mathbf{A}}$ will be often omitted from $s^{\mathbf{A}}$. The circumstance under which s appears will determine if $s \in L$ or s is in fact an interpretation of a symbol of L.

Very often a structure $\mathbf{A}$ is introduced without explicit mention of the related language. But, from the definition of the structure $\mathbf{A}$ it will be clear what is the corresponding language and in that case we shall denote the language in question by $L_{\mathbf{A}}$ and it will be called the language of the model $\mathbf{A}$.

A similar situation may appear for a theory T; the corresponding language will be denoted by L_T and it will be called the language of the theory T.

Assume $L \subseteq L'$ are first-order languages, and let $\mathbf{A}$ be a model of L'. Omitting $s^{\mathbf{A}}$ for $s \in L' \setminus L$ from the model $\mathbf{A}$, we obtain a new model $\mathbf{B}$ of L with the domain $B = A$. In this case, $\mathbf{A}$ is called an expansion of the model $\mathbf{B}$, while $\mathbf{B}$ is called a reduct of the model $\mathbf{A}$. If I and I' are interpretations which determine $\mathbf{B}$ and $\mathbf{A}$, respectively, we see that $I = I'|L$.

Definition 2.6.2. Let $\mathbf{A}$ and $\mathbf{B}$ be models of a language L. Then $\mathbf{B}$ is a submodel of $\mathbf{A}$, if and only if:

- if $B \subseteq A$ and $R \in \mathrm{Rel}_L^k$ then $R^{\mathbf{B}} = R^{\mathbf{A}} \cap B^k$,
- if $F \in \mathrm{Fnc}_L^k$ then $F^{\mathbf{B}} = F^{\mathbf{A}}|B^k$,
- if $c \in \mathrm{Const}_L$ then $c^{\mathbf{B}} = c^{\mathbf{A}}$.

The fact that $\mathbf{B}$ is a submodel of $\mathbf{A}$, we shall denote by $\mathbf{B} \subseteq \mathbf{A}$. For example $(N, +, \cdot, \leqslant, 0, 1) \subseteq (R, +, \cdot, \leqslant, 0, 1)$, but for $Y \subset X$, $Y \neq X$, it is not true that

$$(P(Y), \cup, \cap, ^c, \emptyset, Y) \subseteq (P(X), \cup, \cap, ^c, \emptyset, X).$$

Algebras are special types of models; they are models of a languages L with $\mathrm{Rel}_L = \emptyset$. As in the case of algebras, it is possible to introduce notions of a homomorphism and an isomorphism for models, too.

Definition 2.6.3. Let **A** and **B** be models of a language L, and $f\colon A \to B$. The map f is a homomorphism from **A** into **B**, denoted by $f\colon \mathbf{A} \to \mathbf{B}$, if and only if:

- For $R \in \mathrm{Rel}_L^k$, and all $a_1, a_2, \ldots, a_k \in A$,

  $$R^{\mathbf{A}}(a_1, a_2, \ldots, a_k) \text{ implies } R^{\mathbf{B}}(fa_1, fa_2, \ldots, fa_k).$$

  In this case we say that f is concurrent with relations $R^{\mathbf{A}}$ and $R^{\mathbf{B}}$.

- For $F \in \mathrm{Fnc}_L^k$, and all $a_1, a_2, \ldots, a_k \in A$,

  $$f(F^{\mathbf{A}}(a_1, a_2, \ldots, a_k)) = F^{\mathbf{B}}(fa_1, fa_2, \ldots, fa_k).$$

  In this case we say that f is concurrent with operations $F^{\mathbf{A}}$ and $F^{\mathbf{B}}$.

- For $c \in \mathrm{Const}_L$, $f(c^{\mathbf{A}}) = c^{\mathbf{B}}$.

Similarly to the case of algebraic structures, we have the following classification of homomorphisms:

- f is an embedding, if f is 1–1.
- f is an onto-homomorphism (or epimorphism), if f is onto.
- f is a strong homomorphism, if for every $R \in \mathrm{Rel}_L^k$, and $a_1, a_2, \ldots, a_k \in A$,

  $$R^{\mathbf{A}}(a_1, a_2, \ldots, a_k) \text{ holds iff } R^{\mathbf{B}}(fa_1, fa_2, \ldots, fa_k) \text{ holds.}$$

- f is an isomorphism, if f is 1-1 and a strong epimorphism.
- f is an automorphism, if f is an isomorphism and $\mathbf{A} = \mathbf{B}$.

The set of all the automorphisms of a model **A** is denoted by Aut**A**. It is not difficult to see that Aut**A** is a group under function multiplication; this group will be denoted by **AutA**.

Suppose $f\colon \mathbf{A} \to \mathbf{B}$ is a homomorphism. Then we shall use the following conventions:

- If f is an embedding, we shall say that **A** is embedded into **B**.
- If f is an onto map, we shall say that **B** is a homomorphic image of **A**, and we shall occasionally denote this fact by $\mathbf{B} = f(\mathbf{A})$.
- If f is an isomorphism between models **A** and **B**, then we shall write $f\colon \mathbf{A} \approx \mathbf{B}$. The notation $\mathbf{A} \approx \mathbf{B}$ is used to indicate that there is an isomorphism $f\colon \mathbf{A} \approx \mathbf{B}$, and in this case we shall say that **A** and **B** are isomorphic.

2.7 Satisfaction relation

When introducing syntactical objects of PR1, as the terms, formulas and sentences are, we had in mind certain meanings related to these notions. Alfred Tarski's definition of the satisfaction relation $\models$ determines these ideas precisely.

The introduction of this relation also solves the problem of mathematical truth. Namely, a sentence φ will be true in a structure $\mathbf{A}$, if $\mathbf{A} \models \varphi$. Finally, this formalization of the mathematical truth enables a mathematical analysis of metamathematical notions.

We shall first define the values of the terms in models. Let $\mathbf{A}$ be a model of a first-order language L. A valuation or an assignment of the domain A is every map $\mu\colon \mathrm{Var} \to A$. Hence, valuations assign values to variables. The value of a term $u(x_0, \ldots, x_n) \in \mathrm{Term}_L$ in a model $\mathbf{A}$, denoted by $u^{\mathbf{A}}[\mu]$, is defined by induction on the complexity of terms, assuming that $\mu(v_i) = a_i$, $i \in \omega$.

Let $u \in \mathrm{Term}_L$. If $\mathrm{co}(u) = 0$, we distinguish two cases:

- If u is a variable v_i, then $u^{\mathbf{A}}[\mu] = a_i$.
- If u is a constant symbol c, then $u^{\mathbf{A}}[\mu] = c^{\mathbf{A}}$.

Suppose now $\mathrm{co}(u) = n + 1$, and assume that the values of the terms of the complexity $\leqslant n$ are determined. Then there is $F \in \mathrm{Fnc}_L^k$ such that $u = F(u_1, u_2, \ldots, u_k)$ where $u_1, u_2, \ldots, u_k$ are terms of complexity $\leqslant n$. Then, by definition,

$$u^{\mathbf{A}}[\mu] = F^{\mathbf{A}}(u_1^{\mathbf{A}}[\mu], u_2^{\mathbf{A}}[\mu], \ldots, u_k^{\mathbf{A}}[\mu]).$$

It is also common to write $u^{\mathbf{A}}[a_1, a_2, \ldots, a_r]$ or $u[a_1, a_2, \ldots, a_r]$, or $u(a_1, a_2, \ldots, a_r)$, instead of $u^{\mathbf{A}}[\mu]$, if it is clear which model is in question. Here, r is the number of distinct variables appearing in u.

If $\mathbf{A}$ is a model of a language L, an operation F of domain A is derived if there is $t(x_1, x_2, \ldots, x_n) \in \mathrm{Term}_L$ such that

$$F(a_1, a_2, \ldots, a_n) = t^{\mathbf{A}}(a_1, a_2, \ldots, a_n), \quad a_1, a_2, \ldots, a_n \in A.$$

The following proposition says that homomorphisms of a model remain concurrent with respect to the derived operations.

Theorem 2.7.1. *Let* $\mathbf{A}$ *and* $\mathbf{B}$ *be models of a language* L, *and* $h\colon \mathbf{A} \to \mathbf{B}$ *a homomorphism. Then for every term* $u(x_1, x_2, \ldots, x_n)$ *of* L *and all* $a_1, a_2, \ldots, a_n \in A$ *the following holds*:

$$h(u^{\mathbf{A}}[a_1, a_2, \ldots, a_n]) = u^{\mathbf{B}}[ha_1, ha_2, \ldots, ha_n].$$

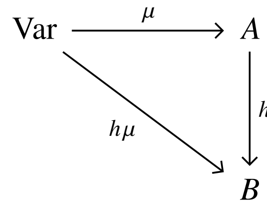


Fig. 2.3 Homomorphism theorem for terms

Proof. The proof is done by induction on the complexity of terms. So, let $u \in \mathrm{Term}_L$, and suppose that the variables $v_0, v_1, \ldots$ have the values $a_0, a_1, \ldots$ under valuation μ. First, assume $\mathrm{co}(u) = 0$, we have two cases:

- $u \in \mathrm{Const}_L$, then: $h(u^{\mathbf{A}}[\mu]) = h(u^{\mathbf{A}}) = u^{\mathbf{B}} = u^{\mathbf{B}}[\mu]$.
- u is a variable x_i, then: $h(u^{\mathbf{A}}[\mu]) = h(a_i) = u^{\mathbf{B}}[ha_1, ha_2, \ldots, ha_n]$.

Now suppose the statement is true for some fixed $n \in \omega$, and let $\mathrm{co}(u) = n + 1$. Then there is an $F \in \mathrm{Fnc}_L^k$ and there are some terms $u_1, u_2, \ldots, u_k$ such that $u = F(u_1, u_2, \ldots, u_k)$. The terms u_i are of complexity $\leqslant n$ and hence, by the inductive hypothesis, we have

$$\begin{aligned} h(u^{\mathbf{A}}[\mu]) &= h(F^{\mathbf{A}}(u_1^{\mathbf{A}}[\mu], \ldots, u_k^{\mathbf{A}}[\mu]) \\ &= F^{\mathbf{B}}(hu_1^{\mathbf{A}}[\mu], \ldots, hu_k^{\mathbf{A}}[\mu]) \\ &= F^{\mathbf{B}}(u_1^{\mathbf{B}}[h\mu], \ldots, u_k^{\mathbf{B}}[h\mu]). \end{aligned}$$

□

Note 2.7.1. This theorem can be obviously restated as follows: For every valuation $\mu \colon \mathrm{Var} \to A$ the diagram in Fig. 2.3 commutes, i.e.,

$$hu^{\mathbf{A}}[\mu] = u^{\mathbf{B}}[h\mu].$$

An algebraic identity of a language L is every formula $u \equiv v$, where $u, v \in \mathrm{Term}_L$. We say that an algebra of L satisfies the identity $u \equiv v$, if and only if for all $a_1, a_2, \ldots, a_n \in A$, $u^{\mathbf{A}}[a_1, a_2, \ldots, a_n] = v^{\mathbf{A}}[a_1, a_2, \ldots, a_n]$.

Corollary 2.7.1. *Let* $\mathbf{A}$ *and* $\mathbf{B}$ *be algebras of a language* L*, and assume that* $\mathbf{B}$ *is a homomorphic image of* $\mathbf{A}$. *Then every identity true in* $\mathbf{A}$ *also holds in* $\mathbf{B}$.

Proof. Let $h\colon A \to B$ be onto, and suppose an identity $u \equiv v$ holds in $\mathbf{A}$. Then, for arbitrary $b_1, b_2, \ldots, b_n \in B$, there are $a_1, a_2, \ldots, a_n \in A$ such that $ha_1 = b_1, \ldots, ha_n = b_n$, So

$$\begin{aligned} u^{\mathbf{B}}[b_1, b_2, \ldots, b_n] &= u^{\mathbf{B}}[ha_1, ha_2, \ldots, ha_n] = hu^{\mathbf{A}}[a_1, a_2, \ldots, a_n] \\ &= hv^{\mathbf{A}}[a_1, a_2, \ldots, a_n] = v^{\mathbf{B}}[ha_1, ha_2, \ldots, ha_n] \\ &= v^{\mathbf{B}}[b_1, b_2, \ldots, b_n]. \end{aligned}$$

□

This corollary is an example of a preservation theorem. Namely, it says that some properties are preserved under homomorphisms and in this case these properties are those which can be described by identities. Some examples are the associativity and the commutativity of algebraic operations. This is probably one of the places where one can see the algebraic nature of model theory.

Now we shall turn to the most important concept of model theory. This is the notion of the satisfaction relation, or the definition of mathematical truth.

Definition 2.7.1. Let $\mathbf{A}$ be a model of a language L. We define the relation $\mathbf{A} \models \varphi[\mu]$ for all formulas φ of L and all valuations $\mu = \langle a_i\colon i \in \omega\rangle$ of the domain A by induction on the complexity of formulas φ:

- If φ is $u \equiv v$, $u, v \in \mathrm{Term}_L$, then

$$\mathbf{A} \models \varphi[\mu] \text{ iff } u^{\mathbf{A}}[\mu] = v^{\mathbf{A}}[\mu].$$

- If φ is $R(u_1, u_2, \ldots, u_n)$, $R \in \mathrm{Rel}_L^n$, $u_1, u_2, \ldots, u_n \in \mathrm{Term}_L$, then

$$\begin{aligned} \mathbf{A} \models \varphi[\mu] \text{ iff } & (u_1^{\mathbf{A}}[\mu], u_2^{\mathbf{A}}[\mu], \ldots, u_n^{\mathbf{A}}[\mu]) \in R^{\mathbf{A}}, \text{ i.e.,} \\ & R^{\mathbf{A}}(u_1^{\mathbf{A}}[\mu], u_2^{\mathbf{A}}[\mu], \ldots, u_n^{\mathbf{A}}[\mu]). \end{aligned}$$

- If φ is $\neg\psi$, then

$$\mathbf{A} \models \varphi[\mu] \text{ iff } \textbf{not } \mathbf{A} \models \psi[\mu].$$

- If φ is $\psi \wedge \theta$, then

$$\mathbf{A} \models \varphi[\mu] \text{ iff } \mathbf{A} \models \psi[\mu] \textbf{ and } \mathbf{A} \models \theta[\mu].$$

- If φ is $\psi \vee \theta$, then

$$\mathbf{A} \models \varphi[\mu] \text{ iff } \mathbf{A} \models \psi[\mu] \textbf{ or } \mathbf{A} \models \theta[\mu].$$

- If φ is $\psi \Rightarrow \theta$, then

$$\mathbf{A} \models \varphi[\mu] \text{ iff } \textbf{not } \mathbf{A} \models \psi[\mu] \textbf{ or } \mathbf{A} \models \theta[\mu].$$

- If φ is $\psi \Leftrightarrow \theta$, then

$$\mathbf{A} \models \varphi[\mu] \text{ iff } \mathbf{A} \models \psi[\mu] \textbf{ if and only if } \mathbf{A} \models \theta[\mu].$$

- If φ is $\exists v_k \psi(v_1, v_2, \ldots, v_n)$, $k \leqslant n$, then

$$\mathbf{A} \models \varphi[\mu] \text{ iff } \textbf{exists } a \in A \textbf{ so that } \mathbf{A} \models \psi[\mu(k/a)].$$

- If φ is $\forall v_k \psi(v_1, v_2, \ldots, v_n)$, $k \leqslant n$, then

$$\mathbf{A} \models \varphi[\mu] \text{ iff } \textbf{for all } a \in A \textbf{ it is valid } \mathbf{A} \models \psi[\mu(k/a)].$$

By the definition of the satisfaction relation, we see that the value of $\mathbf{A} \models \varphi[\mu]$ depends only on the free variables which occur in φ. A rigorous proof of this fact can be derived by induction on the complexity of formulas.

This property enables us to introduce the following conventions: if $\varphi = \varphi(v_0, \ldots, v_n)$ and $\mu = \langle a_i : i \in \omega \rangle$, then we shall simply write $\mathbf{A} \models \varphi[a_0, \ldots, a_n]$ instead of $\mathbf{A} \models \varphi[\mu]$.

Sentences do not have free variables, so their values do not depend on the choice of a valuation, i.e., if $\varphi \in \mathrm{Sent}_L$ and $\mathbf{A} \models \varphi[\mu]$, then for all valuations σ we have $\mathbf{A} \models \varphi[\sigma]$. Thus, we shall use the abbreviated form $\mathbf{A} \models \varphi$ instead of $\mathbf{A} \models \varphi[\mu]$

The theory of a model **A** of L is another useful model-theoretic concept:

$$\mathrm{Th}\mathbf{A} = \{\varphi \in \mathrm{Sent}_L : \mathbf{A} \models \varphi\}$$

It is easy to see that for each formula φ of L and every valuation μ either $\mathbf{A} \models \varphi[\mu]$ or $\mathbf{A} \models \neg\varphi[\mu]$, thus, Th**A** is a complete theory.

For example, the theory of the structure of natural numbers, Th**N**, is complete, and so it is called a complete arithmetic. As **N** is a model of the theory PA, it follows that PA$\subseteq$ Th**N**. On the other hand, the *Gödel's Second Incompleteness Theorem* states that the set of theorems of PA is a proper subset of Th**N**. Moreover, Th**N** is not an axiomatic theory, i.e., it does not have a recursive set of axioms.

One of the tasks of model theory is to solve the problem of whether a given theory is axiomatic.

Let T be a theory of a language L. A model $\mathbf{A}$ of L is a model of the theory T, if every axiom of T holds in $\mathbf{A}$, i.e., $T \subseteq \mathrm{Th}\mathbf{A}$. In such case, we write $\mathbf{A} \models T$.

For example, every ordered field, like an ordered field of rational numbers or real numbers, is a model of theory of ordered fields. Similarly, every Boolean algebra is a model of theory BA.

Every model $\mathbf{A}$ of a language L satisfies all the axioms of the first-order logic (predicate calculus) for L. Rules of inferences (Modus Ponens and Generalization rules) are preserved by the satisfaction relation, i.e., if μ is a valuation of domain A and $\mathbf{A} \models \varphi_1[\mu], \ldots, \varphi_n[\mu]$, where $\varphi_1, \varphi_2, \ldots, \varphi_n \in \mathrm{For}_L$ and ψ is derived by applications of these rules, then $\mathbf{A} \models \psi[\mu]$.

Therefore, the following theorem is easily proved by induction on the length of proofs in T.

Theorem 2.7.2 (Soundness theorem)**.** *Assume $\mathbf{A}$ is a model of a language L and T is a theory of L. If $\mathbf{A} \models T$ and $T \vdash \varphi$, where $\varphi \in \mathrm{Sent}_L$, then $\mathbf{A} \models \varphi$.*

Two models $\mathbf{A}$ and $\mathbf{B}$ of a language L are elementary equivalent if $\mathbf{A}$ and $\mathbf{B}$ satisfy the same sentences of L, i.e., $\mathrm{Th}\mathbf{A} = \mathrm{Th}\mathbf{B}$. This relation between models is denoted by $\mathbf{A} \equiv \mathbf{B}$. It is also said that $\mathbf{A}$ and $\mathbf{B}$ have the same first-order properties.

By induction on the complexity of formulas it is easy to show:

Theorem 2.7.3. *Let $g\colon \mathbf{A} \approx \mathbf{B}$ be an isomorphism of models $\mathbf{A}$ and $\mathbf{B}$ of a language L. Then, for every formula $\varphi v_0 \ldots v_n$ of L and every valuation $\mu = \langle a_i \colon i \in \omega \rangle$ of the domain A, the following holds*:

$$\mathbf{A} \models \varphi[a_0, \ldots, a_n] \ \textit{if and only if} \ \mathbf{B} \models \varphi[ga_0, \ldots, ga_n].$$

Since the value of a sentence in a model does not depend on the choice of a valuation, we have the following consequence.

Corollary 2.7.2. *If $\mathbf{A}$ and $\mathbf{B}$ are isomorphic models of a language L, then $\mathbf{A} \equiv \mathbf{B}$.*

Therefore isomorphisms preserve first-order properties.

Elementary embeddings of models are embeddings which preserve first-order properties. Hence, an elementary embedding between models $\mathbf{A}$ and $\mathbf{B}$ of a language L is every map $g\colon A \to B$, such that for all $\varphi \in \mathrm{For}_L$ and all valuations of domain A, it satisfies

$$\mathbf{A} \models \varphi[a_0, \ldots, a_n] \ \text{if and only if} \ \mathbf{B} \models \varphi[ga_0, \ldots, ga_n].$$

In this case we use the notation $g\colon \mathbf{A} \overset{\prec}{\longrightarrow} \mathbf{B}$.

If $\mathbf{A} \subseteq \mathbf{B}$ and the inclusion map $i_A\colon \mathbf{A} \to \mathbf{B}$, $i_A\colon x \mapsto x$, $x \in A$, is elementary, then we write $\mathbf{A} \prec \mathbf{B}$. Observe that $\mathbf{A} \prec \mathbf{B}$ implies $\mathbf{A} \equiv \mathbf{B}$.

A class $\mathfrak{M}$ of models of a language L is axiomatic if there is a theory T of L such that $\mathfrak{M} = \{\mathbf{A} \colon \mathbf{A} \models T\}$. For example, the class of all ordered fields is axiomatic and so is the class of all Boolean algebras.

The class of all cyclic groups is not an axiomatic class. Also, if a theory T has infinitely many non-isomorphic finite models, then the class of all finite models of T is not an axiomatic class.

The class of all models of a theory T is denoted by $\mathfrak{M}(T)$. The central theorem of model theory says:

Theorem 2.7.4. *For every consistent theory T, $\mathfrak{M}(T) \neq \emptyset$.*

2.8 Method of new constants

Introduction of new linguistic constants is a dual procedure to the process of interpretations. Namely, to every nonempty set A there corresponds a certain language L_A:

- If R is a k-ary relation over A, then let $\underline{R}$ be a relation symbol of length k which belongs to L_A.
- If g is an n-ary operation over domain A, we can introduce a function symbol $\underline{g} \in L_A$ of arity k.
- If $a \in A$ then $\underline{a} \in \mathrm{Const}_{L_A}$.

The symbols $\underline{R}$, $\underline{g}$, $\underline{a}$ are called names of R, g, a, respectively. We have a natural interpretation of the language L_A so defined:

If $s \in L_A$, then $\underline{s}^A = s$.

In this way we have built a model $\mathbf{A} = (A, \mathcal{R}, \mathcal{F}, C)$, where $\mathcal{R}$ is the set of all relations over A, $\mathcal{F}$ is the set of all operations with domain A, and $C = A$.

It is not always necessary to consider the full expansion of set A. For example, if $\mathbf{A}$ is any model of a language L, and $a_1, a_2, \ldots, a_n \in A$, then $\mathbf{A}' = (A, a_1, \ldots, a_n)$ is a simple expansion of $\mathbf{A}$, and $\mathbf{A}'$ is a model of the language $L' = L \cup \{\underline{a}_1, \ldots, \underline{a}_n\}$. Note that $\varphi\underline{a}_1\underline{a}_2 \ldots \underline{a}_n$ is a sentence of $L \cup \{\underline{a}_1, \underline{a}_2, \ldots, \underline{a}_n\}$.

The following proposition is interesting for two reasons. The first one relates to the inductive nature of the satisfaction class. Secondly, this proposition shows that the satisfaction relation can be defined only for sentences if the starting model is modified.

Theorem 2.8.1 (Satisfaction relation theorem on sentences)**.** *Let* **A** *be a model of a language* L *and* $\varphi v_0 v_1 v_2 \dots v_n \in \mathrm{For}_L$*. Then, for all* $a_0, a_1, a_2, \dots, a_n \in A$*, we have*

$$\mathbf{A} \models \varphi[a_0, a_1, a_2, \dots, a_n] \quad \textit{iff} \quad (\mathbf{A}, \underline{a}_0, \underline{a}_1, \underline{a}_2, \dots, \underline{a}_n) \models \varphi[\underline{a}_0, \underline{a}_1, \underline{a}_2, \dots, \underline{a}_n]. \tag{2.8.1}$$

Proof of this theorem is simple, but long and tedious:

Step 1 If $t(v_0, v_1, v_2, \dots, v_n) \in \mathrm{Term}_L$, $\mathbf{A}' = (\mathbf{A}, a_0, a_1, a_2, \dots, a_n)$, by induction on the complexity of the terms one proves:

$$t^{\mathbf{A}'}(\underline{a}_0, \underline{a}_1, \underline{a}_2, \dots, \underline{a}_n) = t^{\mathbf{A}}[a_0, a_1, a_2, \dots, a_n].$$

Step 2 By induction on the complexity of formulas one proves (2.8.1).

For example we prove the induction step $\varphi = \exists v_i \varphi$. We take $i = 0$, $\varphi = \varphi v_1 v_2 \dots v_n$ and $\psi = \psi(v_0, v_1, v_2, \dots, v_n)$. Then

$$\begin{aligned}
\mathbf{A} \models \varphi[a_1, a_2, \dots, a_n] \quad &\text{iff} \quad \text{for some } b \in A, \quad \mathbf{A} \models \psi[b, a_1, a_2, \dots, a_n] \\
&\phantom{\text{iff}} \quad \text{using inductive hypothesis} \\
&\text{iff} \quad \text{for some } b \in A, \quad (\mathbf{A}', b) \models \psi \underline{b} \underline{a}_1 \underline{a}_2 \dots \underline{a}_n \\
&\text{iff} \quad \text{for some } b \in A, \quad \mathbf{A}' \models \theta[b] \\
&\phantom{\text{iff}} \quad \text{where } \theta x = \psi x \underline{a}_1 \underline{a}_2 \dots \underline{a}_n, \text{ so} \\
&\text{iff} \quad \mathbf{A}' \models \exists x \theta x \\
&\text{iff} \quad \mathbf{A}' \models \varphi \underline{a}_1 \underline{a}_2 \dots \underline{a}_n
\end{aligned}$$

We shall apply the previous proposition in the following theorem which says that there is no quite satisfactory model theory for finite structures. The reason is that the relation of elementary equivalence and the isomorphisms of models coincide for finite structures.

Theorem 2.8.2. *Let* **A** *and* **B** *be models of a language* L*. If* A *is finite and* $\mathbf{A} \equiv \mathbf{B}$*, then* $\mathbf{A} \approx \mathbf{B}$*.*

Proof. Assume $|A| = n$. Then $\mathbf{A} \models \sigma_n$, where σ_n = "There are exactly n elements". But **A** and **B** are elementary equivalent, so $B \models \sigma_n$. Therefore, **A** and **B** have the same number of elements.

Now we prove the following fact:

Claim 2.8.1. *If* **A** *and* **B** *are finite models and* $\mathbf{A} \equiv \mathbf{B}$*, then for each* $a \in A$ *there is a* $b \in B$ *such that* $(\mathbf{A}, a) \equiv (\mathbf{B}, b)$*.*

Proof of Claim. Suppose $a \in A$ and let $B = \{b_1, b_2, \dots, b_n\}$. Assume there is no $b \in B$ such that $(\mathbf{A}, a) \equiv (\mathbf{B}, b)$, and choose a constant symbol $c \notin L$, the so-called *new constant symbol.*

Then, for all $i \leqslant n$ there is a formula $\varphi_i x$ of the language L and there is $b_i \in B$ such that $(\mathbf{A}, a) \models \varphi_i c$ and $(\mathbf{B}, b_i) \models \neg\varphi_i c$, where c is interpreted by a in (A, a) and by b_i in $(\mathbf{B}, b_i)$. Hence,

$$(\mathbf{A}, a) \models \bigwedge_{j \leqslant n} \varphi_j c, \text{ so } \mathbf{A} \models \exists x \bigwedge_{j \leqslant n} \varphi_j x.$$

Since $\mathbf{A} \equiv \mathbf{B}$, we have $\mathbf{B} \models \exists x \bigwedge_{j \leqslant n} \varphi_j x$. Thus for some $k \leqslant n$, $\mathbf{B} \models \bigwedge_{j \leqslant n} \varphi_j[b_k]$. By previous theorem it follows that $(\mathbf{B}, b_k) \models \bigwedge_{j \leqslant n} \varphi_j \underline{b}_k$, hence, $(\mathbf{B}, b_k) \models \bigwedge_{j \leqslant n} \varphi_j c$ if c is interpreted by b_k. This is a contradiction to the choice of the formula φ_k. □

By repeated use of Claim, we can find an enumeration $a_1, a_2, \ldots, a_n$ of domain A, so that $(\mathbf{A}, a_1, a_2, \ldots, a_n) \equiv (\mathbf{B}, b_1, b_2, \ldots, b_n)$, where $(\mathbf{A}, a_1, a_2, \ldots, a_n)$ and $(\mathbf{B}, b_1, b_2, \ldots, b_n)$ are models of a language $L \cup \{c_1, c_2, \ldots, c_n\}$. Then the map $f\colon A \to B$ defined by $f\colon a_i \mapsto b_i$, $i \leqslant n$, is an isomorphism of models $\mathbf{A}$ and $\mathbf{B}$.

For example, if $*$ is a binary operation symbol of a language L, then for some $a_i, a_j, a_k \in A$ that satisfy $a_k = a_i *^{\mathbf{A}} a_j$ we have that

$$(\mathbf{A}, a_1, a_2, \ldots, a_n) \models \underline{a}_k = \underline{a}_i * \underline{a}_j, \text{ so } (\mathbf{B}, b_1, b_2, \ldots, b_n) \models \underline{b}_k = \underline{b}_i * \underline{b}_j.$$

Hence $b_k = b_i *^{\mathbf{B}} b_j$.

Therefore we proved $f(a_i *^{\mathbf{A}} a_j) = f(a_i) *^{\mathbf{B}} f(a_j)$, i.e., f is concurrent in respect to the operations $*^{\mathbf{A}}$ and $*^{\mathbf{B}}$.

In a similar way one can show that f is concurrent with relations of models $\mathbf{A}$ and $\mathbf{B}$. Obviously, f is onto. This map is also 1–1, since

$$(\mathbf{A}, a_1, a_2, \ldots, a_n) \models \underline{a}_i \equiv \underline{a}_j \quad \text{iff} \quad (\mathbf{B}, b_1, b_2, \ldots, b_n) \models \underline{b}_i \equiv \underline{b}_j.$$

Thus $f\colon \mathbf{A} \approx \mathbf{B}$. □

The idea of constructing an isomorphism as in the previous theorem is often used in model theory. It is summarized as follows.

Theorem 2.8.3. *Let* $\mathbf{A}$ *and* $\mathbf{B}$ *be models of a language* L, $A = \{a_i : i \in I\}$, $B = \{b_i : i \in I\}$, *and let* $\mathbf{A}' = (\mathbf{A}, a_i)_{i \in I}$, $\mathbf{B}' = (\mathbf{B}, b_i)_{i \in I}$ *be models of a language* $L \cup \{c_i : i \in I\}$ *with* c_i *interpreted in* $\mathbf{A}'$ *by* a_i *and in* $\mathbf{B}'$ *by* b_i. *Then,*

$$(\mathbf{A}, a_i)_{i \in I} \equiv (\mathbf{B}, b_i)_{i \in I} \quad \textit{implies} \quad \mathbf{A} \approx \mathbf{B}.$$

As expected, $f\colon \mathbf{A} \approx \mathbf{B}$ where $f\colon a_i \mapsto b_i$, $i \in I$.

Chapter 3

Algebraic and combinatorial constructions

In this chapter we discuss two topics of algebraic and combinatorial nature. The first one concerns group actions on models and related notions, such as orbits and class equations. We will use this constructs in counting special types of partial orders and lattices.

The second topic is related to the so called inversion formulas, an important branch of combinatorics. We use this theory in finding particular types of automorphisms of models and permutations of their domains, such as having a certain number of fixed points. This section contains an original contributions such as extensions of Gould inversion formulas. Also, we believe that our approach is novel as it is based on Hopf algebras. Most results in this section are published in the paper [31].

3.1 Group action on a model

In our work an important role will have the notion of group action on a model. Several concepts are related to group action, such as orbits, stabilizers, class equation, etc. For example, we can measure symmetry of a structure by the number of orbits. In the extreme case we have a highly symmetric structure with only one orbit. Such structures have transitive group action. We shall see that in some way symmetry of a structure is inverse to the definability in a structure.

Let $\mathbf{A}$ be a model of L and $G = \mathrm{Aut}(\mathbf{A})$, i.e., G is a group of automorphisms of a model $\mathbf{A}$. We define a natural action $\theta\colon G \to \mathrm{Sym}(A)$, by $\theta(g) = g$, or in other words $(\theta(g))(a) = g(a)$. Obviously θ is an embedding of G into $\mathrm{Sym}(A)$.

We define in respect to θ the following relation on a domain A:

$$a \sim b \text{ iff } \bigvee_{g \in G} ga = b$$

It is easy to see that this relation is an equivalence relation on a domain A. Very often we write a^g instead of ga for $g \in G$, $a \in A$.

The equivalence class $[a] = \{x \in A \colon x \sim a\}$ of an element $a \in A$ is called an orbit and it is denoted by O_a. Observe that

$$O_a = \{a^g \colon g \in G\} = a^G.$$

We can choose a transversal $T \subseteq A$, such that T meets each orbit in exactly one element. Then $\mathscr{P} = \{O_a \colon a \in T\}$ is a partition of A and we have the so called class equation

$$|A| = \sum_{a \in T} |O_a|.$$

The notion of a stabilizer G_a of an element $a \in A$ will have an important role in further discussion. It is defined in the following way

$$G_a = \{g \in G \colon a^g = a\}.$$

Obviously G_a is a subgroup of G.

Theorem 3.1.1. $|O_a| = |G \colon G_a|$

Proof. Let $G/G_a = \{gG_a \colon g \in G\}$. Then for $g, h \in G$ we have

$$\begin{aligned} a^g = a^h &\Leftrightarrow g(a) = h(a) \\ &\Leftrightarrow g^{-1}h(a) = a \\ &\Leftrightarrow g^{-1}h \in G_a \\ &\Leftrightarrow g^{-1}hG_a = G_a \\ &\Leftrightarrow hG_a = gG_a \end{aligned}$$

Hence $\lambda \colon O_a \to G/G_a$, defined by $\lambda \colon a^g \to gG_a$, $g \in G$, is a well-defined, 1–1 and onto map. □

Corollary 3.1.1 (Class equation 2nd form)**.** *From the 1st form of class equation we have*

$$|A| = \sum_{a \in T} |G/G_a|.$$

Example 3.1.1. Let $B = \mathbf{2}^n$ be a Boolean algebra. It is well-known that automorphisms of B are generated by permutations of atoms. Hence $\mathrm{Aut}(B) = S_n$. B is partitioned into $n + 1$ levels L_k. Each level L_k consists of elements having exactly k atoms below them eventually including themselves. For example, $L_0 = \{\mathbf{0}\}$, $L_1 = \{a \in B\colon a \text{ is an atom}\}$, $L_2 = \{a \in B\colon \text{there is exactly 2 atoms below } a\}$, etc. If $a \in L_i, b \in L_j$ and $i \neq j$, then there is no automorphism that sends a to b, since any automorphism must preserve the number of atoms below a. On the other hand, suppose $a, b \in L_k$ and let $c_1, \ldots, c_p$ be the common atoms below both a and b, further $a_1, \ldots, a_q$ atoms that are below a but not b, and $b_1, \ldots, b_q$ atoms that are below b but not a. Observe that $p + q = k$. Let g be an automorphism generated by the permutation of atoms which fixes c_i and sends a_i to b_i. Then $ga = b$ and so a and b belong to the same orbit. Therefore we proved that orbits under natural action of B are exactly the levels L_k.

Example 3.1.2. Let G be a complete graph over a domain with n elements. Then $\mathrm{Aut}(G) = S_n$ and there is only one orbit under the natural action, the whole G.

Example 3.1.3. Let L_n be the linear order of a set with n elements. Then $\mathrm{Aut}(L_n) = I$, where I is identical mapping. Hence, orbits are exactly one element subsets of L_n.

3.2 Inversion formulas

The main idea of this section is explained by the following example. Let C be the set of complex numbers, Z the set of integers, N the set of nonnegative integers and $\mathbf{C}^Z$ the complex vector space. We remind that a function $\pi_n\colon C^Z \to C$, $n \in Z$, is a projection if $\pi_n(f) = f(n)$, $f \in C^Z$. Let A be the subspace of the vector space of linear functionals of $\mathbf{C}^Z$ where A is generated by projections π_n, $n \in Z$. We introduce an associative and commutative algebra $\mathcal{A} = (A, \cdot)$ over A defining multiplication of projections by $\pi_m \cdot \pi_n = \pi_{m+n}$. Obviously, the power $(\pi_1)^n$ is equal to π_n. Hence, if π denotes π_1, then $\pi^n = \pi_n$. By projection calculus we shall mean calculation in the algebra $\mathcal{A}$. It appears that the algebra $\mathcal{A}$ is very appropriate for computing various inversion formulas from discrete mathematics.

Now we proceed to our example. Let $\mathcal{F} = \{f \in C^Z\colon \bigwedge_{n<0} f(n) = 0\}$. Obviously $\mathcal{F}$ is a subspace of $\mathbf{C}^Z$ and we may identify $\mathcal{F}$ and $\mathbf{C}^N$. Also, $f \in \mathcal{F}$ if and only if for all $n < 0$, $\pi^n(f) = \pi_n(f) = 0$. We shall prove by projection calculus the following well known inversion formula

$$g_n = \sum_k \binom{n}{k} f_{n-2k} \quad \Leftrightarrow \quad f_n = \sum_{2k \leqslant n} (-1)^k \frac{n}{n-k} \binom{n-k}{k} g_{n-2k}, \qquad f, g \in \mathcal{F}. \tag{3.2.1}$$

For this purpose let us introduce the following functional $\theta = \pi + \pi^{-1}$. Then

$$\theta^n = \sum_k \binom{n}{k} \pi^{n-2k}. \tag{3.2.2}$$

If $f \in \mathcal{F}$ and $g \in F$ is defined by $g_n = g(n) = \theta^n(f)$, $n \in N$, then by (3.2.2) we have $g_n = \sum_k \binom{n}{k} f_{n-2k}$. For the proof of the equivalence (3.2.1), we express π^n by a polynomial of θ using Tchebychev polynomials. Let $T_n(x)$ be Tchebychev polynomial of the first kind and $C_n(x) = \frac{1}{2}T_n(2x)$. Then $C_n(x)$ is also called Tchebychev polynomial of the first kind and it is well known $C_n(x)$ satisfies the following identities (see for example [1] or [26]):

$$C_n(x + x^{-1}) = x^n + x^{-n}, \quad n \geqslant 0, \tag{3.2.3}$$

$$C_n(x) = \sum_{2k \leqslant n} (-1)^k \frac{n}{n-k} \binom{n-k}{k} x^{n-2k}, \quad n > 0. \tag{3.2.4}$$

Hence, we have

$$\pi^n = C_n(\theta) - \pi^{-n}, \quad n \in Z. \tag{3.2.5}$$

Therefore, if $f \in \mathcal{F}$ and $n > 0$ then, as $\pi^{-n}(f) = 0$, we have $f_n = \pi^n(f) = C_n(\theta)(f) - \pi^n(f) = C_n(\theta)(f)$. Hence

$$f_n = \sum_{2k \leqslant n} (-1)^k \frac{n}{n-k} \binom{n-k}{k} \theta^{n-2k}(f) = \sum_{2k \leqslant n} (-1)^k \frac{n}{n-k} \binom{n-k}{k} g_{n-2k}.$$

Thus we proved direction ($\Rightarrow$) of equivalence (3.2.1). The other direction follows from the following observation. The equalities $g_m = \sum_k \binom{m}{k} f_{m-2k}$, $m = 0, 1, \ldots, n$, can be written as $P \cdot F = G$, where P is a regular triangular matrix, $G = [g_0, g_1, \ldots, g_n]$ and $F = [f_0, f_1, \ldots, f_n]$. Then the righthand side of the equivalence (3.2.1) is written as $F = Q \cdot G$ where $Q = P^{-1}$ and entries of Q are exactly coefficients appearing in expansion of f_n by g_{n-2k}. As from $F = Q \cdot G$ follows $P \cdot F = G$, it also follows the direction ($\Leftarrow$) in (3.2.1).

3.2.1 Hopf algebra of projection functions

We show that the algebra $\mathcal{A}$ discussed in the previous section and similar algebras naturally bear the structure of Hopf algebra. Even if the next definitions and analysis can be applied to an arbitrary field $\mathbf{K}$, we shall assume $\mathbf{K} = \mathbf{C}$. Let $I \subseteq C$, $I \neq \emptyset$ and A the subspace of the vector space of linear functionals of the complex space $\mathbf{C}^I$, generated by projection functions π_i, $i \in I$. Suppose that I is a subgroup of the additive group of $\mathbf{C}$ or of the multiplicative part C^* of $\mathbf{C}$.

Assuming the usual notation for Hopf algebras and related notions (see for example [11] or [39]), it is easy to see that in both of the following two cases we obtain a Hopf algebra.

Additive case, I is a subgroup of $(C, +, 0)$. Hopf algebra $H_I = (A, \nabla, \mathbf{1}, \triangle, \varepsilon)$ over the complex field $\mathbf{C}$ is defined as follows: $\nabla(\pi_i \otimes \pi_j) = \pi_{i+j}$, in multiplicative notation $\pi_i \cdot \pi_j = \pi_{i+j}$, $\mathbf{1}(z) = \pi_0$, $z \in C$, $\triangle(\pi_i) = \pi_i \otimes \pi_i$ and $\varepsilon(\pi_i) = 0$, $i, j \in I$. The map $a\colon \pi_i \mapsto \pi_{-i}$, $i \in I$, is the antipod.

Multiplicative case, I is a subgroup of $(C^*, \cdot, 1)$. Hopf algebra $H_I = (A, \nabla, \mathbf{1}, \triangle, \varepsilon)$ over the complex field $\mathbf{C}$ is defined taking: $\nabla(\pi_i \otimes \pi_j) = \pi_{ij}$, in multiplicative notation $\pi_i \cdot \pi_j = \pi_{ij}$, $\mathbf{1}(z) = \pi_1$, $z \in C$, $\triangle\ (\pi_i) = \pi_i \otimes \pi_i$ and $\varepsilon(\pi_i) = 1$, $i, j \in I$. The map $a\colon \pi_i \mapsto \pi_{i^{-1}}$, $i \in I$, is the antipod.

Obviously, in both cases H_I is commutative and in fact H_I is a Hopf subalgebra of the dual Hopf algebra of the group Hopf algebra $\mathbf{C}[I]$. If $I = (Z, +, 0)$ we obtain algebra $\mathcal{A}$ presented in the previous section.

Even if the additive and the multiplicative cases are similarly defined, they may produce examples of quite different nature. In additive case if we take $\pi = \pi_1$, we may write π^i instead of π_i and if I is the set of real numbers, then the ring $\boldsymbol{A}_I = (A, +, \cdot, \mathbf{0}, \mathbf{1})$ is an integral domain and it is isomorphic to the ring of posynomials over C in the variable π, see [12] and [32]. If I is the additive group of integers, then $\boldsymbol{A}_I = \mathbb{Z}[\pi, \pi^{-1}]$ is the ring of Laurent polynomials in the indeterminate π. On the other hand, in the multiplicative case if $I = \langle \varepsilon \rangle$, $\varepsilon^n = 1$, then $\boldsymbol{A}_I$ has divisors of zero. For example, if ε is a primitive root of $x^3 - 1$ and $a = \mathbf{1} + \pi_\varepsilon + \pi_{\varepsilon^2}$, $b = \mathbf{1} + \varepsilon^2 \pi_\varepsilon + \varepsilon \pi_{\varepsilon^2}$, then $ab = \mathbf{0}$.

Let $S \subseteq I$ and $S' = I \setminus S$. Then we can identify the space $\mathbf{C}^S$ with the subspace $\mathcal{F}_S = \{f \in C^Z \colon \bigwedge_{x \in S'} f(x) = 0\}$ of $\mathbf{C}^I$. In the example from the previous section, obviously S is the set of nonnegative integers. For deriving inversion formulas for functions $f\colon S \to C$, we shall use their replicas in $\mathcal{F}_S$. This derivation is related but not the same as that one in the umbral calculus, see for example [9]. We also note that Tchebychev polynomial and their variants have been already the subject of investigation in context of Hopf algebras and from the purely algebraic point of view, see for example [5] and [13].

3.2.2 Linear functional $\theta = \pi + \pi^{-r}$

Let us suppose notation and definitions as previously introduced. Here we shall consider linear functionals of $\mathbf{C}^Z$ of the form

$$\theta = \pi + \pi^{-r}, \quad r \text{ is a nonnegative integer.} \tag{3.2.6}$$

in the ring $\boldsymbol{A}_I = \mathbb{Z}[\pi, \pi^{-1}]$. Then for $m = r + 1$

$$\theta^n = \sum_k \binom{n}{k} \pi^{n-mk}, \quad n = 1, 2, \ldots. \tag{3.2.7}$$

We shall prove that π^n can be expressed as stated in the following theorem.

Theorem 3.2.1. *There are polynomials $S_n(x)$ and $Q_n(x)$ with integer coefficients such that*

$$\pi^n = S_n(\theta) - Q_n(\pi^{-1}), \quad Q_n(0) = 0. \tag{3.2.8}$$

Our main aim is to find explicitly polynomials $S_n(x)$ and $Q_n(x)$. For this purpose, we shall need some properties of symmetric functions related to the following polynomial

$$p(x) = x^m + ax^{m-1} + b, \quad a, b \in C. \tag{3.2.9}$$

Let $\lambda_1, \lambda_2, \ldots, \lambda_m$ be the roots of the polynomial p(x) and $s_n(a, b)$ the nth power sum of the roots

$$s_n(a, b) = \lambda_1^n + \lambda_2^n + \cdots + \lambda_m^n. \tag{3.2.10}$$

Using Girard-Waring formula for symmetric functions, Gould (see [18] or [25]) derived the following formula

$$s_n(a, b) = \sum_{0 \leqslant k \leqslant \frac{n}{m}} (-1)^{n-rk} \frac{n}{n - rk} \binom{n - rk}{k} a^{n-mk} b^k, \quad m = r + 1. \tag{3.2.11}$$

Using this formula, the following proposition is easily deduced.

Proposition 3.2.1. *Let $m = r + 1$ and $u_n = s_n(-a, b)$, where $s_n(a, b)$ is defined by (3.2.11). Then for all positive integers n*

$$u_n = \sum_{0 \leqslant k \leqslant \frac{n}{m}} (-1)^k \frac{n}{n - rk} \binom{n - rk}{k} a^{n-mk} b^k. \tag{3.2.12}$$

If $1 \leqslant n \leqslant r$ then $u_n = a^n$ and also $u_m = a^m - mb$. The sequence u_n with these initial conditions is the unique solution of the difference equation

$$v_{n+1} - av_n + bv_{n-r} = 0. \tag{3.2.13}$$

Proof. The identity (3.2.12) immediately follows from (3.2.11). For the second part of the proposition, we can write (3.2.12) as

$$u_n = a^n + \sum_{1\leqslant k\leqslant \frac{n}{m}} (-1)^k \frac{n}{n-rk}\binom{n-rk}{k} a^{n-mk} b^k. \qquad (3.2.14)$$

If $n \leqslant r$ then the sum in (3.2.14) is empty, hence $u_n = a^n$. Similarly,

$$u_m = a^m - mb + \sum_{2\leqslant k\leqslant \frac{m}{m}} (-1)^k \frac{m}{m-rk}\binom{m-rk}{k} a^{m-mk} b^k = a^m - mb. \qquad (3.2.15)$$

The sequence u_n satisfies the recurrence (3.2.13) since u_n is a linear combination of the roots of the characteristic equation of (3.2.13). The order of this recurrence is m and values for $u_1, u_2, \ldots, u_m$ are determined, hence the uniqueness follows. □

Now we proceed to the proof of the identity (3.2.8). For this purpose, we shall deliver recurrence relations for the polynomials $S_n(\theta)$ and $Q_n(\pi)$.

Lemma 3.2.1. *Let r be a nonnegative integer, $m = r + 1$, $\theta = \pi + \pi^{-r}$ and $1 \leqslant l \leqslant r$. Then for the polynomials in* (3.2.8) *we can take*:

(*i*) $S_l(\theta) = \theta^l$ *and* $S_m = \theta^m - m$.

(*ii*) $Q_l(t) = t^{rl}((1 + t^{-m})^l - t^{-ml})$, $Q_m(t) = t^{rm}((1 + t^{-m})^m - t^{-m^2} - mt^{m-m^2})$.

Proof. By identity (3.2.7), after short calculation we have

$$\pi^l = \theta^l - \sum_{k\geqslant 1}\binom{l}{k}\pi^{l-mk} = \theta^l - \pi^{-rl}((1+\pi^m)^l - \pi^{ml}). \qquad (3.2.16)$$

Hence, $S_l(\theta) = \theta^l$ and $Q_l(\pi^{-1}) = \pi^{-rl}((1 + \pi^m)^l - \pi^{ml})$ for $1 \leqslant l \leqslant r$. Taking $t = \pi^{-1}$, we have $Q_l(t) = t^{rl}((1 + t^{-m})^l - t^{-ml})$.

By identity (3.2.7) we also have

$$\pi^m = \theta^m - m - \sum_{k\geqslant 2}\binom{m}{k}\pi^{m-mk}, \qquad (3.2.17)$$

hence

$$\pi^m = (\theta^m - m) - \pi^{-rm}((1+\pi^m)^m - \pi^{m^2} - m\pi^{m^2-m}). \qquad (3.2.18)$$

Therefore $S_m = \theta^n - m$ and $Q_m(\pi^{-1}) = \pi^{-rm}((1 + \pi^m)^m - \pi^{m^2} - m\pi^{m^2-m})$. Taking $t = \pi^{-1}$ we have $Q_m(t) = t^{rm}((1 + t^{-m})^m - t^{-m^2} - mt^{-m^2+m})$. □

Now we shall deliver the recursive relations for polynomials $S_n(t)$ and $Q_n(t)$ appearing in (3.2.8). In the following we shall take $m = r + 1$.

Assuming the identity (3.2.8), we have $\theta\pi^n = \theta S_n(\theta) - \theta Q_n(\pi^{-1})$, hence

$$\begin{aligned}(\pi + \pi^{-r})\pi^n &= \theta S_n(\theta) - (\pi + \pi^{-r})Q_n(\pi^{-1}),\\ \pi^{n+1} &= \theta S_n(\theta) - \pi^{n-r} - (\pi + \pi^{-r})Q_n(\pi^{-1}).\end{aligned} \tag{3.2.19}$$

Assuming the recurrence (3.2.8) for $n - r$, i.e., $\pi^{n-r} = S_{n-r}(\theta) - Q_{n-r}(\pi^{-1})$, we obtain

$$\pi^{n+1} = \theta S_n(\theta) - S_{n-r}(\theta) - ((\pi + \pi^{-r})Q_n(\pi^{-1}) - Q_{n-r}(\pi^{-1})). \tag{3.2.20}$$

By (3.2.8) we have $\pi^{n+1} = S_{n+1}(\theta) - Q_{n+1}(\pi^{-1})$ and comparing with (3.2.20), we have

$$\begin{aligned}S_{n+1}(\theta) &= \theta S_n(\theta) - S_{n-r}(\theta),\\ Q_{n+1}(\pi^{-1}) &= (\pi + \pi^{-r})Q_n(\pi^{-1}) - Q_{n-r}(\pi^{-1}).\end{aligned} \tag{3.2.21}$$

Taking the substitution $t = \pi^{-1}$ and using (3.2.21) it is easy to deduce that $Q_n(t)$ satisfies the recurrence

$$tQ_{n+1}(t) = (1 + t^{r+1})Q_n(t) - tQ_{n-r}(t). \tag{3.2.22}$$

Lemma 3.2.2. *$Q_n(0) = 0$ for all $n > 0$.*

Proof. Let $m = r + 1$. First assume $1 \leqslant n \leqslant r$. By Lema 3.2.1 we have

$$\begin{aligned}Q_n(t) &= t^{rn}((1 + t^{-m})^n - t^{-mn})\\ &= t^{rn}\left(1 + \binom{n}{1}t^{-m} + \cdots + \binom{n}{n-1}t^{-m(n-1)}\right).\end{aligned}$$

As $rn - m(n - 1) = m - n > 0$, it follows that $t \mid Q_n(t)$. Further,

$$\begin{aligned}Q_m(t) &= t^{rm}((1 + t^{-m})^m - t^{-m^2} - mt^{-m^2+m})\\ &= t^{rm}\left(1 + \binom{m}{1}t^{-m} + \cdots + \binom{m}{m-2}t^{-m(m-2)}\right).\end{aligned}$$

As $-m^2 + 2m + rm = m > 0$, it follows that $t \mid Q_n(t)$.

Therefore, we proved that $t \mid Q_n(t)$ for $n \leqslant m$, i.e., $Q_n(0) = 0$. For $n > m$ we use the recurrence (3.2.22). We see immediately that $t \mid Q_n(t)$. □

Corollary 3.2.1. *The constant term of $Q_n(t)$ is equal to* 0.

Proof of Theorem 3.2.1. The proof immediately follows by induction from the recurrence relations (3.2.21), derivations (3.2.19), (3.2.20) and the previous corollary. □

Now we deliver the explicit forms of the polynomials $S_n(x)$ and $Q_n(t)$.

Proposition 3.2.2. *Let r be a positive integer, $m = r + 1$ and assume a sequence of polynomials $S_n(x) \in \mathbb{Z}[x]$ satisfies:*

(*i*) $S_{n+1}(x) = xS_n(x) - S_{n-r}(x)$,

(*ii*) $S_l(x) = x^l$, $1 \leqslant l \leqslant r$, *and* $S_m(x) = x^m - m$.

Then for all positive integers n

$$S_n(x) = \sum_{0 \leqslant k \leqslant \frac{n}{m}} (-1)^k \frac{n}{n-rk} \binom{n-rk}{k} x^{n-mk}. \tag{3.2.23}$$

Proof. The characteristic equation of the recurrence (i) is:

$$\lambda^m - x\lambda^{m-1} + 1 = 0. \tag{3.2.24}$$

It is easy to see that the equation (3.2.24) has no multiple roots, hence if $\lambda_1, \lambda_2, \ldots, \lambda_m$ are roots of (3.2.24) then

$$S_n(x) = c_1\lambda_1^n + c_2\lambda_2^n + \cdots + c_m\lambda_m^n, \tag{3.2.25}$$

for some unique constants $c_1, c_2, \ldots, c_m$. As any linear combination of the n-th powers of $\lambda_1, \lambda_2, \ldots, \lambda_m$ satisfies the recurrence (3.2.23), due to the initial conditions (ii) and the uniqueness of the constants $c_1, c_2, \ldots, c_m$, by Proposition 3.2.1 we have $c_i = 1$, $i \leqslant m$, and

$$S_n(x) = s_n(-x, 1) = \sum_{0 \leqslant k \leqslant \frac{n}{m}} (-1)^k \frac{n}{n-rk} \binom{n-rk}{k} x^{n-mk}. \tag{3.2.26}$$

□

We note that for the most applications the explicit form (3.2.26) of the polynomials $S_n(t)$ and the recurrence (3.2.8) are sufficient. However, if one wants to find the polynomials $Q_n(t)$, it is possible and convenient to introduce new polynomials $R_n(t)$ and $h_n(t)$ with integer coefficients related to $Q_n(t)$ in the following way:

$$\begin{aligned} Q_n(t^{-1}) &= t^{-rn}R_n(t), \quad n \in N, \\ R_n(t) &= h_n(t^m), \quad n \in N. \end{aligned} \tag{3.2.27}$$

We also note that the polynomials $S_n(x)$ are related to the so called incomplete polynomials, see [36], and to the orthogonal polynomials on the radial rays in the complex plane which were introduced by G. Milovanović in [34] and studied in details in [35].

It is easy to prove the following proposition.

Proposition 3.2.3. *Let r be a nonnegative integer, $m = r+1$, $\theta = \pi + \pi^{-r}$ and $1 \leqslant l \leqslant r$. Then:*

(i) $R_l(\pi) = (1+\pi^m)^l - \pi^{ml}$, $R_m(\pi) = (1+\pi^m)^m - \pi^{m^2} - m\pi^{m^2-m}$.

(ii) $\deg(R_n(\pi)) < rn$, $n \in N$.

(iii) $h_{n+1}(t) = (1+t)h_n(x) - t^r h_{n-r}(x)$.

(iv) $h_l(x) = (1+t)^l - t^l$, $1 \leqslant l \leqslant r$, *and* $h_m(x) = (1+t)^m - t^m - mt^{m-1}$.

From the next theorem and relations (3.2.27) immediately follows the explicit form of the polynomial $Q_n(t)$.

Theorem 3.2.2. *Assume a sequence of polynomials $h_n(t) \in \mathbb{Z}[t]$ satisfies conditions (iii) and (iv) in the previous proposition. Then for all positive integers n*

$$h_n(t) = f_n(t) + g_n(t), \tag{3.2.28}$$

where

$$f_n(t) = (t+1)^n - t^n, \quad g_n(t) = \sum_{1 \leqslant k \leqslant \frac{n}{m}} (-1)^k \frac{n}{n-rk} \binom{n-rk}{k} (1+t)^{n-mk} t^{rk}. \tag{3.2.29}$$

Proof. The characteristic equation of the recurrence (i) is

$$\lambda^m - (1+t)\lambda^{m-1} + t^r = 0. \tag{3.2.30}$$

The roots $\lambda_1, \lambda_2, \ldots, \lambda_m$ of this equation are distinct and it's one root is t, so we can take $\lambda_{r+1} = \lambda_m = t$. Let $u_n = \lambda_1^n + \lambda_2^n + \cdots + \lambda_r^n$ and $s_n = u_n + \lambda_m^n = u_n + t^n$. From Proposition 3.2.1, $s_n = (1+t)^n$ for $1 \leqslant n \leqslant r$ and $s_m = (1+t)^m - mt^{m-1}$. Hence, $u_n = (1+t)^n - t^n$ for $1 \leqslant n \leqslant r$ and $u_m = (1+t)^m - t^m - mt^{m-1}$. Also, u_n obviously satisfies the recurrence $u_{n+1}(t) = (1+t)u_n(t) - t^r u_{n-r}(t)$. Therefore, due to the same initial conditions for u_n and h_n, by induction we have immediately $u_n = h_n$ for all n, i.e., $h_n(t) = s_n - t^n$. By Proposition 3.2.1, we have

$$s_n = \sum_{0 \leqslant k \leqslant \frac{n}{m}} (-1)^k \frac{n}{n-rk} \binom{n-rk}{k} (1+t)^{n-mk} t^{rk}. \tag{3.2.31}$$

wherefrom we immediately deliver (3.2.28). □

3.2.3 Gould inversion formula

As an application of the projection calculus and the operator θ introduced in the previous section we prove Gould inversion formula, see [16] or [17]. This formula is a generalization of inversion

formula (3.2.1) and it states

$$g_n = \sum_k \binom{n}{k} f_{n-mk} \quad \Leftrightarrow \quad f_n = \sum_{0 \leqslant k \leqslant \frac{n}{m}} (-1)^k \frac{n}{n-rk} \binom{n-rk}{k} g_{n-mk}, \tag{3.2.32}$$

where $f, g \in \mathcal{F}$, r is a nonnegative integer and $m = r + 1$. For the proof we use the same technique as in the case $r = 1$ where we used projection calculus, Tchebychev polynomials and their crucial properties (3.2.3) and (3.2.4). By use of this technique, the proof of Gould formula directly follows from Theorem 3.2.1 and the explicit form (3.2.26) of the polynomial $S_n(x)$.

Using linear functional identities (3.2.7) and (3.2.1) and explicit forms of the polynomials $S_n(\theta)$ and $Q_n(\pi^{-1})$, we can generalize Gould inversion formula. Namely, we can find new inversion formulas for

$$g_n = \sum_k \binom{n}{k} f_{n-mk+l}, \quad 0 \leqslant l < m. \tag{3.2.33}$$

It is particularly simple to deliver the inversion formula for the case $m = 2$. To see this, assume $m = 2$ and let us introduce the new linear functional $\sigma_n = \pi\theta^n$. Then $\sigma_n = \sum_k \binom{n}{k} \pi^{n-2k+1}$. By (3.2.5), i.e., inversion formula $\pi^n = C_n(\theta) - \pi^{-n}$, we have $\pi^{n+1} = \pi C_n(\theta) - \pi^{1-n}$ and so

$$\pi^{n+1} = \sum_k (-1)^k \frac{n}{n-k} \binom{n-k}{k} \sigma_{n-2k+1} - \pi^{1-n}. \tag{3.2.34}$$

Hence we obtain the following inversion formula for (3.2.33), case $m = 2, l = 1$:

$$f_{n+1} = \sum (-1)^k \frac{n}{n-k} \binom{n-k}{k} g_{n-2k+1} - f_{1-n}. \tag{3.2.35}$$

We note that for a given sequence $g \in \mathcal{F}$, the sequence f is not uniquely determined by (3.2.35) as indices in g_i are shifted by one:

$$g_0 = f_1,\ g_1 = f_2 + f_0,\ g_2 = f_3 + 2f_1, \ldots \tag{3.2.36}$$

while

$$f_1 = g_0,\ f_2 = g_1 - f_0,\ f_3 = g_2 - 2g_0, \ldots \tag{3.2.37}$$

where f_0 is arbitrary. Also note that f_{1-n} vanishes for $n > 1$. In a similar manner we can obtain the inversion formula for (3.2.33) in the general case. So assume (3.2.33) and let us introduce

the linear functional $\sigma_n = \pi^l \theta^n$. By Theorem 3.2.1 and Proposition 3.2.2 we have

$$\begin{aligned} \pi^{n+l} &= \pi^l S_n(\theta) - \pi^l Q_n(\pi^{-1}) \\ &= \sum_{0\leqslant k\leqslant \frac{n}{m}} (-1)^k \frac{n}{n-rk}\binom{n-rk}{k}\sigma_{n-mk+l} - \pi^l Q_n(\pi^{-1}). \end{aligned} \tag{3.2.38}$$

By (3.2.27), Lemma 3.2.2 and as $Q_n(0) = 0$

$$Q_n(t^{-1}) = t^{-rn} h_n(t^m) = c_n t^{-\lambda_n} + c_n' t^{-\lambda_n - m} + c_n'' t^{-\lambda_n - 2m} + \cdots \tag{3.2.39}$$

where $1 \leqslant \lambda_n \leqslant m$. Hence

$$\pi^l Q_n(\pi^{-1}) = c_n \pi^{l-\lambda_n} + H_n(\pi^{-1}) \tag{3.2.40}$$

for some polynomial $H_n(t)$ with integer coefficients. Therefore we obtain the inversion formula for (3.2.33), $0 \leqslant l < m$:

$$f_{n+l} = \sum_{0\leqslant k\leqslant \frac{n}{m}} (-1)^k \frac{n}{n-rk}\binom{n-rk}{k} g_{n-mk+l} - c_n f_{l-\lambda_n}. \tag{3.2.41}$$

We see that $f_0, f_1, \dots, f_l$ can be chosen arbitrarily. The coefficient c_n can be obtained from the representation of the polynomial h_n given by Theorem 3.2.2. Here we shall find the power λ_n:

Proposition 3.2.4.

$$\lambda_n = m - \rho_m(n) \tag{3.2.42}$$

where $\rho_m(n)$ is the remainder of division of n by m (remainder function).

Proof. Note that $m = r + 1$. As $Q_n(t) = t^{rn} h_n(t^{-m})$ and $Q_n(0) = 0$, for powers t^{rn-lm} of terms in $Q_n(t)$ we have $rn - lm > 0$, so $l < rn/m$. First assume $m \mid n$. Then for the smallest power $t^{\lambda_n} = t^{rn-\bar{l}n}$ in $Q_n(t)$ we have $\bar{l} = rn/m - 1$, so $\lambda_n = rn - \bar{l}m = m = m - \rho_m(n)$. Assume $m \nmid n$. Then for $\bar{l} = [rn/m]$ we have $\bar{l} = [n - n/m] = n - 1 - [n/m]$, so $\lambda_n = rn - \bar{l}m = m - \rho_m(n)$. □

In the similar way we can deliver various inversion formulas such as appearing in [23] by studying associated functionals in the Hopf algebra $\mathcal{A}$. For example, for delivering the inverse formula for $g_n = f_n + f_{n-1} + f_{n-2}$, see [19], one may use the functional $\theta = \pi + \pi^{-1} + \pi^{-2}$.

3.2.4 Example

We infer a formula for the number $\sigma_{n,m}$ of permutations with exactly m fixed points over a set having n elements. For this we use the following inversion formula:

$$u_n = \sum_{k=0}^{n} \binom{n}{k} v_{n-k} \Leftrightarrow v_n = \sum_{k=0}^{n} (-1)^k \binom{n}{k} u_{n-k}. \tag{3.2.43}$$

Derivation. Let S_n denote the set of all permutations over $\{1, 2, \ldots, n\}$. Further, let $u_n = n! = |S_n|$ and

$$v_n = \text{the number of permutations over } \{1, 2, \ldots, n\} \text{ not having a fixed point.}$$

Let $\mathcal{P}_m = \{p \in S_n\colon p \text{ has exactly } m \text{ fixed points}\}$. If $p \in \mathcal{P}_m$ and $a_1, \ldots, a_m$ are fixed points of p, then the restriction $p \upharpoonright \{1, 2, \ldots, n\} \setminus \{a_1, \ldots, a_m\}$ is a permutation over a set with $n - m$ elements without fixed points. Hence $\sigma_{n,m} = |\mathcal{P}_m| = \binom{n}{m} v_{n-m}$.

$\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_n$ is a partition of S_n, hence as $S_n = \bigcup_{k=0}^{n} \mathcal{P}_k$,

$$\begin{aligned} |S_n| &= \sum_{k=0}^{n} |\mathcal{P}_k|, \quad \text{i.e.,} \\ n! &= \sum_{k=0}^{n} \binom{n}{k} v_{n-k}, \quad \text{so by (3.2.43)} \\ v_n &= \sum_{k=0}^{n} (-1)^k \binom{n}{k} (n-k)! = n! \sum_{k=0}^{n} \frac{(-1)^k}{k!}. \end{aligned}$$

Therefore,

$$\sigma_{n,m} = \binom{n}{m} v_{n-m} = \frac{n!}{m!(n-m)!}(n-m)! \sum_{k=0}^{n-m} \frac{(-1)^k}{k!},$$

$$\sigma_{n,m} = \frac{n!}{m!} \sum_{k=0}^{n-m} \frac{(-1)^k}{k!}. \tag{3.2.44}$$

Remark 3.2.1. For large n, $e^{-1} \approx \sum \frac{(-1)^k}{k!}$, so $v_n \approx n! e^{-1}$.

Let us consider this example for some specific values, e.g., $n = 7$ and $m = 3$. Then:

(i) According to the remark the approximate value of $\sigma_{7,3}$ is

$$\sigma_{7,3} \approx \frac{7!}{3!e} \approx \frac{5040}{6e} \approx 309.$$

(ii) Using the summation formula (3.2.44) we can obtain the exact value

$$\sigma_{7,3} = 315.$$

(iii) The self explanatory code in Listing 3.1 uses our system to explicitly compute the number of permutations with exactly three fixed points over the domain having seven elements. As we see the result agrees with the result of computation in (ii).

Here is a brief explanation of this code:

- Lines 7–10 are preamble that imports necessary building blocks for use of the system from Python.
- Line 15 defines the computation domain `S`.
- Lines 21, 24, 27, and 34 describe in first-order logic permutations g having exactly three fixed points. In the predicate calculus they are:

$$\begin{gathered}\forall i \in S \; \exists j \in S \; g(i) = j \\ \forall i \in S \; \forall k, l \in D \; g(i) = k \Rightarrow g(i) \neq l \\ \forall k \in S \; \forall i, j \in D \; g(i) = k \Rightarrow g(j) \neq k \\ \exists i, j, k \in S \, (i \neq j \wedge i \neq k \wedge j \neq k \wedge g(i) = i \wedge g(j) = j \wedge g(k) = k \wedge \\ \forall l \in S \, (g(l) = l \Rightarrow l = i \vee l = j \vee l = k))\end{gathered} \tag{3.2.45}$$

  In the program we used a suitable set `V` of quadruplets of distinct elements of `S` in order to reduce the size of formula (3.2.45).
- As can be seen on line 37, we chose to count models on the GPU. For counting models on the CPU, a `CPURunner` should be used instead.

Note 3.2.1. We could omit formula `f2` since `f1` and `f3` imply `f2`. However, in this case, the program will execute more slowly. The reason is that additional constraints help to speed up reductions.

Note 3.2.2. The quantifiers in `f4` are not ordinary bounded first-order quantifiers, but a kind of "differential" quantifiers. In other words they are a kind of branching or Henkin quantifiers.

```
'''
This program counts the number of permutations g over S = {0,...,n-1}
with exactly 3 fixed points. Permutations are encoded by propositional
variables p(i,j) in the following way:
  g(i) = j iff p(i,j) = 1.
'''
from logic import bool
from logic.abc import p, i, j, k, l
from logic.fo.quantifiers import *
from logic.cl.runner2 import GPURunner

n = 7

# Definition of set S
S = range(n)

# Definition of set D of ordered pairs (a,b) in SxS so that a < b
D = comb(S,2)

# Statement: domain of g is S
f1 = A[i:S].E[j:S] (p(i,j))

# Statement: g is a function
f2 = A[i:S].A[k,l:D] (~p(i,k) | ~p(i,l))

# Statement: g is 1-1
f3 = A[k:S].A[i,j:D] (~p(i,k) | ~p(j,k))

# Definition of set V of ordered quadruplets (a,b,c,d) in S^4
# where a, b, c, and d are mutually distinct elements
V = perm(S,4)

# Statement: g has exactly three fixed points
f4 = E[i,j,k:V].A[l] (p(i,i) & p(j,j) & p(k,k) & ~p(l,l))

# Result
print(GPURunner().count(bool.Expression(f1 & f2 & f3 & f4)))
```

Listing 3.1 Permutations$(n, 3)$

Namely, the choice of values of variables under the scope of quantifiers, like $\exists i, j, k \in D$ are triplets of elements which are mutually distinct. This follows from the structure of set D, which consists of quartets $Q = (q_1, q_2, q_3, q_4)$ of mutually distinct elements. In the above bounded quantifier, i takes the first coordinate q_1, j takes the second coordinate q_2, and k takes the third coordinate q_3. Therefore, i, j, k are mutually distinct. Further, in quantifier part $\forall l \in D$, the choice of value for l depends on previously selected values for i, j, k like in Henkin quantifiers, since l takes the coordinate q_4. This implementation of quantifiers in our system enables us to write much simpler formulas than it would be possible in ordinary first-order predicate calculus. It should be noted that the branching property of quantifiers was obtained by choice of the domain set D. This allows us further enhancements in solution of the stated example. By the nature of the problem we could take that D consists of quartets where $q_1 < q_2 < q_3$, while q_4 is an arbitrary element in the complement $S \setminus \{q_1, q_2, q_3\}$. These enhancements retain the simple structure of the formulas `f1`, ..., `f4` that describe the problem, but significantly reduce the execution time.

Chapter 4

Parallel model counting

4.1 Introduction to model counting

Finite model counting is the classical counting counterpart (in the sense of algorithmic complexity) of the Boolean satisfiability problem or SAT. It generalizes SAT and is in fact the canonical #P-complete problem[1], which makes it highly useful and extremely difficult to solve in practice.

Effective model counting procedures would open up a range of new applications. For example, various probabilistic inference problems, such as Bayesian net reasoning, can be effectively translated into model counting problems. Another application is in the study of hard combinatorial problems, such as combinatorial designs, the number of automorphisms of finite structures, some problems in graph theory and various partition problems over finite sets.

In general, there are two main approaches to exact model counting [4]. The first one is based on DPLL[2]-style exhaustive search. Model counters of this type are essentially modified SAT solvers like Relsat [3] and Cachet [37]. By contrast, the second approach is based on "knowledge compilation", i.e., conversion of the formula into a special normal form with certain desirable properties. An example of this kind of compiler is c2d [10], which converts the given CNF formula into deterministic, decomposable negation normal form or d-DNNF. Although the resulting normal form allows us to count models very efficiently, the compiler still uses a SAT solver under the hood. So we can say that the model counters are, in a way, based on SAT solvers. And given that the parallelization of SAT solvers was met with the limited success [20][24], it is not surprising that the model counters are basically sequential in nature as well.

[1]The complexity class #P (pronounced "number P" or "sharp P") consists of all counting problems associated with the decision problems in the set NP. Unlike the most well-known complexity classes, it is not a class of decision problems but a class of function problems. It was first defined in [41].

[2]Davis-Putnam-Logemann-Loveland

On the other hand, the amount of computational resources in modern hardware has progressed tremendously over the past decade. Therefore, we thought that it would be a good idea to develop a model counter capable of utilizing all of the available hardware resources. Particularly, we had in mind easily accessible GPUs, which can nowadays have thousands of computing cores. And that is the primary contribution of this research: design and implementation of a software system for model counting on modern GPUs.

The whole system is envisioned as a library for Python programming language, which will provide GPU accelerated primitives and other basic blocks for building model counters tailored to specific problems. This customizability of the system will prove to be one of its greatest strengths.

In the second section we provide the mathematical background of the parallelized algorithm for the evaluation of Boolean expressions on which we based the system. The basic idea is that if we let $f(x_1, x_2, \dots, x_n)$ be a Boolean expression in n variables $x_1, x_2, \dots, x_n$, then we can construct n Boolean vectors $b_1, b_2, \dots, b_n$ of size 2^n with the following property:

($\mathscr{P}$) $f(b_1, b_2, \dots, b_n)$ is a Boolean vector that codes the full DNF of f.

We show that the vectors $b_1, b_2, \dots, b_n$ are exactly the free generators of a free Boolean algebra with n free generators. Furthermore, we explain how to use them to parallelize the computation of f. The idea of parallelization of computing logical operations in this way is indicated in [30].

In the third section we give a formal description of a translation procedure from first-order predicate formulas to propositional formulas, which we implement in the system. This idea is formally developed in [33], but it was used in the study of problems in the infinitary combinatorics, particularly in finding their complexity in the Borel hierarchy.

Finally, in the fourth section we provide the implementation details such as specific algorithms and data structures used to realise the essential functionality of the system.

Standard notation and terminology from model theory is assumed as in [7] and [29]. Also, for notions from universal algebras we shall refer to [6]. Models of a first order language L are denoted by bold capital letters $\mathbf{A}$, $\mathbf{B}$, etc, while their domains are represented respectively by A, B and so on. By a domain we mean any nonempty set. The letter L will be used to denote a first-order language. The first order logic is denoted by $L_{\omega\omega}$ and the propositional calculus with a set $\mathcal{P}$ of propositional variables by $L_{\omega}^{\mathcal{P}}$, or simply L_{ω}. The set of natural numbers $\{0, 1, 2, \dots\}$ is denoted by N. We also take $2 = \{0, 1\}$. By $\mathbf{2}$ we denote the two-element Boolean algebra and then $\mathbf{2}^I$ is the power of $\mathbf{2}$, while $\mathbf{0}$ and $\mathbf{1}$ are respectively the smallest and the greatest element of $\mathbf{2}^I$. Occasionally elements of $\mathbf{2}^I$ are called Boolean vectors. Whenever is needed to distinguish the formal equality sign from identity, for the first one we shall keep $=$, while $\equiv$ will denote identity.

4.2 Variables

In this section we explain the logical and algebraic background of our computing method. The power of a model $\mathbf{A}$, the product $\prod_{i\in I}\mathbf{A}$, is denoted by $\mathbf{A}^I$.

4.2.1 Interpretation of variables

By a set of variables we mean any nonempty set V so that no $v \in V$ is a finite sequence of other elements from V. This assumption secures the unique readability of terms and formulas. Particularly we shall consider finite and countable sets of variables V, e.g. $V = \{v_0, v_1, \dots\}$. A valuation of a domain A is any map from V to A. Let I denote the set of all valuations from domain A, i.e., $I = A^V$. In this section, the letter I will be reserved for the set of valuation of a domain A. Sometimes we shall assume that elements from I will have finite supports.

Definition 4.2.1. (*Interpretation of variables*). Let v be a variable from V. The interpretation of the variable v in the domain A is the map $\hat{v}\colon I \to A$ defined by $\hat{v}(\mu) = \mu(v)$, $\mu \in I$.

The set of interpretations of variables from V into domain $\mathbf{A}$ is denoted by $\hat{V}_A$. Therefore, $\hat{V}_A = \{\hat{v}\colon v \in V\}$.

Let $\varphi(v_1, \dots, v_n)$ be a formula of a language L having free variables $v_1, \dots, v_n$ and let $\mathbf{A}$ be a model of L. The map $\hat{\varphi}^{\mathbf{A}}(\hat{v}_1, \dots, \hat{v}_n)$, abbreviated by $\hat{\varphi}^{\mathbf{A}}$, is $\hat{\varphi}^{\mathbf{A}}\colon I \to 2$ defined by $\hat{\varphi}^{\mathbf{A}}(\mu) = 1$ if $\mathbf{A} \models \varphi[\mu]$, otherwise $\hat{\varphi}^{\mathbf{A}}(\mu) = 0$, $\mu \in I$. Hence $\hat{\varphi}^{\mathbf{A}} \in 2^I$.

Proposition 4.2.1. *Let φ be an identity $s = t$, where s and t are terms of L. Then the following are equivalent*:

$$1^\circ\ \mathbf{A}^I \models \varphi[\hat{v}_1, \dots, \hat{v}_n], \quad 2^\circ\ \hat{\varphi}^{\mathbf{A}}(\hat{v}_1, \dots, \hat{v}_n) = \mathbf{1}, \quad 3^\circ\ \mathbf{A} \models \varphi[\mu],\ \mu \in I. \tag{4.2.1}$$

Proof. The equivalence of 2° and 3° follows immediately by definition 4.2.1. From 3° follows 1° since identities are preserved under products of models. Finally, assume 1°. Then

$$s^{\mathbf{A}^I}(\hat{v}_1, \dots, \hat{v}_n) = t^{\mathbf{A}^I}(\hat{v}_1, \dots, \hat{v}_n). \tag{4.2.2}$$

Let $\pi_\mu\colon \mathbf{A}^I \to \mathbf{A}$ be a projection, $\mu \in I$. Since π_μ is a homomorphism we have

$$\begin{aligned} \pi_\mu(s^{\mathbf{A}^I}(\hat{v}_1, \dots, \hat{v}_n)) &= s^{\mathbf{A}}(\pi_\mu\hat{v}_1, \dots, \pi_\mu\hat{v}_n) \\ &= s^{\mathbf{A}}(\hat{v}_1(\mu), \dots, \hat{v}_n(\mu)) \\ &= s^{\mathbf{A}}(\mu(v_1), \dots, \mu(v_n)) = s^{\mathbf{A}}[\mu]. \end{aligned} \tag{4.2.3}$$

Hence, 3° follows by 4.2.2. □

For an algebra $\mathbf{A}$ of L let $\mathcal{J}(\mathbf{A})$ be the set of all identities that are true in $\mathbf{A}$. Similarly, $\mathcal{J}(\mathcal{K})$ denotes the set of all identities that are true in all algebras of a class $\mathcal{K}$ of algebras of L. If $\mathcal{J}(\mathbf{A}) = \mathcal{J}(\mathbf{B})$, $\mathbf{A}$ and $\mathbf{B}$ are algebras of L, we shall also write $\mathbf{A} \equiv_{\mathcal{J}} \mathbf{B}$.

The notion of interpretation of variables will play the fundamental role in our analysis and program implementation. They can be useful in other cases, too. For example, using this notion it is easy to prove the famous Birkhoff theorem on the existence of free algebras and other related theorems such as Birkhoff HSP theorem.

Theorem 4.2.1. (*G. Birkhoff*) *Let $\mathcal{K}$ be a nontrivial abstract*[3] *class of algebras of L, closed under subalgebras and products. Then $\mathcal{K}$ has a free algebra over every nonempty set.*

Proof. It easy to see, for example by use of downward Skolem-Löwenheim theorem, that for each algebra $\mathbf{A} \in \mathcal{K}$ there is at most countable subalgebra $\mathbf{A}'$ of $\mathbf{A}$ so that $\mathbf{A}' \equiv_{\mathcal{J}} \mathbf{A}$. The algebra $\mathbf{A}'$ is obviously isomorphic to an algebra which domain is a subset of N. Hence, there is a set $\mathcal{K}' = \{\mathbf{A}_s : s \in S\}$ of at most countable algebras such that $\mathcal{K}' \subseteq \mathcal{K}$ and $\mathcal{J}(\mathcal{K}) = \mathcal{J}(\mathcal{K}')$.

Let $\mathbf{A} = \prod_s \mathbf{A}_s$ be the product of all algebras from $\mathcal{K}'$. Since $\mathcal{K}$ is closed under products, it follows $\mathbf{A} \in \mathcal{K}$, hence $\mathcal{J}(\mathcal{K}) \subseteq \mathcal{K}(\mathbf{A})$. On the other hand, for each $s \in S$, $\mathbf{A}_s$ is a homomorphic image of $\mathbf{A}$, as $\mathbf{A}_s = \pi_s \mathbf{B}$. Hence each identity φ of L which holds on $\mathbf{A}$ is also true in all algebras from $\mathcal{K}'$ and therefore in all algebras from $\mathcal{K}$. So we proved

$$\mathcal{J}(\mathcal{K}) = \mathcal{J}(\mathbf{A}). \tag{4.2.4}$$

Since $\mathcal{K}$ is nontrivial, it must be $|A| \geqslant 2$. Let X be any nonempty set. For our purpose we may identify X with $\hat{V}_A$ for some set of variables V. Let $\mathbf{\Omega}$ be subalgebra of $\mathbf{A}^I$ generated by $\hat{V}_A$. Now we prove that $\mathbf{\Omega}$ is the free algebra over $\hat{V}_A$ for class $\mathcal{K}$. Let $\mathbf{B} \in \mathcal{K}$ be an arbitrary algebra and $g : \hat{V}_A \to B$. Each element $a \in \Omega$ is of the form $a = s^{\mathbf{\Omega}}(\hat{v}_1, \ldots, \hat{v}_n)$ for some L-term s and some (different) variables $v_1, \ldots, v_n \in V$. We extend g to $f : \ : \mathbf{\Omega} \to \mathbf{B}$ taking

$$f(a) = s^{\mathbf{B}}(g\hat{v}_1, \ldots, g\hat{v}_n). \tag{4.2.5}$$

The map f is well defined. Indeed, suppose that for some other term t of L, $a = t^{\mathbf{\Omega}}(\hat{v}_1, \ldots, \hat{v}_n)$. Let φ denote the identity $s(v_1, \ldots, v_n) = t(v_1, \ldots, v_n)$. Then $s^{\mathbf{\Omega}}(\hat{v}_1, \ldots, \hat{v}_n) = t^{\mathbf{\Omega}}(\hat{v}_1, \ldots, \hat{v}_n)$ and as $\mathbf{\Omega} \subseteq \mathbf{A}^I$ it follows $\mathbf{A}^I \models \varphi[\hat{v}_1, \ldots, \hat{v}_n]$. By Proposition 4.2.1 it follows that the identity φ holds on $\mathbf{A}$. By (4.2.4) then φ is true in all algebras from $\mathcal{K}$. Hence

$$s^{\mathbf{B}}(g\hat{v}_1, \ldots, g\hat{v}_n) = t^{\mathbf{B}}(g\hat{v}_1, \ldots, g\hat{v}_n), \tag{4.2.6}$$

and thus we proved that the f is well-defined.

[3] closed for isomorphic images

In a similar manner we prove that f is a homomorphism. For simplicity, suppose $*$ is a binary operation of L. We denote the interpretations of $*$ in $\mathbf{\Omega}$ and $\mathbf{B}$ by $\cdot$. Take $a, b \in \Omega$ and let s and t be terms of L so that

$$a = s^{\mathbf{\Omega}}(g\hat{v}_1, \dots, g\hat{v}_n), \quad b = t^{\mathbf{\Omega}}(g\hat{v}_1, \dots, g\hat{v}_n) \tag{4.2.7}$$

and let w be the combined term $w = s * t$. Then

$$f(a \cdot b) = f(w^{\mathbf{\Omega}}(\hat{v}_1, \dots, \hat{v}_n)) = w^{\mathbf{B}}(g\hat{v}_1, \dots, g\hat{v}_n) = g(a) \cdot g(b). \tag{4.2.8}$$

Thus, f is a homomorphism from $\mathbf{\Omega}$ to $\mathbf{B}$ which extends g. □

The novelty of this proof in respect to the standard proof is that it does not use the notion of a term algebra (absolutely free algebra) and that it has the model-theoretic nature.

Suppose $\mathcal{K}$ is the class of algebras to which the previous theorem refers. We note the following.

Note 4.2.1.1 Assume $\mathbf{A} \in \mathcal{K}$ is an arbitrary algebra which satisfies condition

$$\mathcal{J}(\mathcal{K}) = \mathcal{J}(\mathbf{A}).$$

Such an algebra $\mathbf{A}$ will be called the characteristic algebra for the class $\mathcal{K}$. By close inspection of our proof of Theorem 4.2.1, we noted that this condition suffices to construct the free algebra for $\mathcal{K}$ from $\mathbf{A}$. This idea is indicated to some extent in [6], (Part II, chapter 11, particularly see problem 11.5, p. 77) but under stronger and amended assumptions (it is assumed that $\mathcal{K}$ is a variety generated by $\mathbf{A}$) and without referring to variable interpretations.

4.2.2 Free Boolean vectors

It is well known that finite free Boolean algebras with n free generators are the algebras $\mathbf{2}^{2^n}$. We remark that this immediately follows by note 4.2.1.1, since $\mathbf{2}$ is the characteristic algebra for the class of all Boolean algebras. The structure and properties of free Boolean vectors of $\mathbf{\Omega}_n = \mathbf{2}^{2^n}$ is discussed in [30] in details.

We remind that a collection $\{b_1, \dots, b_n\}$ of elements of a Boolean algebra $\mathbf{B}$ is independent if $b_1^{\alpha_1} \wedge \dots \wedge b_n^{\alpha_n} \neq 0$, where $b^1 = b$ and $b^0 = b'$ (b' denotes the complement of b). A similar definition of independence is for families of subsets of a given set. A collection $\{b_1, \dots, b_n\}$ generates the free subalgebra of $\mathbf{B}$ if and only if it is independent, cf. [38]. In fact, the following holds.

Theorem 4.2.2. *Let $S = \{1, 2, \dots, 2^n\}$ and a_n, b_n, c_n be the sequences defined as follows.*

(*i*) a_n = *number of labeled Boolean algebras with domain* S (*number of different Boolean algebras with domain* S).

(*ii*) b_n = *number of independent collections* $\{P_1, \ldots, P_n\}$ *of subsets of* S.

(*iii*) c_n = *number of free generating sets* $\{b_1, \ldots, b_n\}$ *of* $\mathbf{\Omega}_n$.

Then $a_n = b_n = c_n = (2^n)!/n!$.

Proof. The number of labelings of a finite model $\mathbf{A}$ of size m is equal to $m!/|\mathrm{Aut}(\mathbf{A})|$. As $\mathrm{Aut}(\mathbf{2}^n)$ is isomorphic to the permutation group S_n, it follows $a_n = (2^n)!/n!$.

Let $\mathbf{B} = \mathbf{2}^n$ and $\mathbf{B}^l$ a labeled algebra obtained from $\mathbf{B}$. Algebra $\mathbf{B}$ has exactly n ultrafilters and so has $\mathbf{B}^l$. Let $U(\mathbf{B})$ be the set of all ultrafilters of $\mathbf{B}$. By Theorem 2.2.7 in [30], $U(\mathbf{B})$ is an independent collection of subsets of S. The map U which assigns $U(\mathbf{B}^l)$ to $\mathbf{B}^l$ is $1-1$. Indeed, let us for $S_1, \ldots, S_n \subseteq S$ and $\alpha \in 2^n$ define

$$S^\alpha = S^{\alpha_1} \cap \ldots \cap S^{\alpha_n}. \tag{4.2.9}$$

For $a \in S$ let $P_1, \ldots, P_k, P_{k+1}, \ldots, P_n \in U(\mathbf{B}^l)$ so that $a \in P_1, \ldots, P_k$ and $a \notin P_{k+1}, \ldots, P_n$. Then $P_1 \cap \ldots, \cap P_k \cap P_{k+1}^c \ldots \cap P_n^c = \{a\}$.

Therefore, we proved that for each $a \in B^l$ there is a unique $\alpha \in 2^n$ so that $P^\alpha = \{a\}$, $P_1, \ldots, P_n \in U(\mathbf{B}^l)$. Let $\wedge^l$ and ${}'^l$ be Boolean operations of $\mathbf{B}^l$. Then for $a, b \in B^l$ and corresponding $\alpha, \beta \in 2^n$ we have

$$P^{\alpha'} = \{a^{'l}\}, \quad P^{\alpha \wedge \beta} = \{a \wedge^l b\} \tag{4.2.10}$$

where α', $\alpha \wedge \beta$ are computed in $\mathbf{2}^n$. Thus, we proved that $U(\mathbf{B}^l)$ uniquely determines $\mathbf{B}^l$, hence $a_n \leqslant b_n$.

Suppose $P = \{P_1, \ldots, P_n\}$ is an independent collection of subsets of S. Then P can serve as $U(\mathbf{B}^l)$ for certain labeled Boolean algebra $\mathbf{B}^l$. To prove it, note that each P^α has at least one element and that $\bigcup_{\alpha \in 2^n} P^\alpha$ has at most 2^n elements. This shows that P^α is one-element set. Therefore, by 4.2.10, a Boolean algebra $\mathbf{B}^l$ with domain S is defined and it is easy to see that $P = U(\mathbf{B}^l)$. Hence $a_n = b_n$.

Finally, as noted, a collection $X = \{X_1, \ldots, X_n\}$ of subsets of S freely generates the power set algebra $P(S)$ if and only if X is independent. Hence, $c_n = b_n$. □

We will be dealing particularly with free generators of $\mathbf{\Omega}_n$ of the following form. Let a_i, $i = 0, 1, \ldots, 2^n - 1$, be binary expansions of integers i with zeros padded to the left up to the length n. Let M be the matrix whose columns are vectors a_i. As noted in [30], binary vectors

b_i, $i = 1, 2 \dots n$, formed by rows of M are free vectors of $\mathbf{\Omega}_n$. In case n = 3, the matrix M and vectors b_i are

$$M = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}, \tag{4.2.11}$$

$b_1 = 00001111$, $b_2 = 00110011$, $b_3 = 01010101$.

4.2.3 Computing Boolean expressions

Let $t = t(v_1, \dots, v_n)$ be a Boolean expression in variables $v_1, \dots, v_n$ and $b_1, \dots, b_n$ free generators of $\mathbf{\Omega}_n$.

Proposition 4.2.2. *$t^{\mathbf{\Omega}_n}(b_1, \dots, b_n)$ codes the full DNF of t.*

Proof. By our previous discussion, we may take $b_i = \hat{v}_i$ and $I = \{\hat{v}_1, \dots, \hat{v}_n\}$. Let π_μ be a projection from $\mathbf{\Omega}_n$ to $\mathbf{2}$, $\mu \in I$, and $d = t^{\mathbf{\Omega}_n}(b_1, \dots, b_n)$. Then

$$\pi_\mu d = \pi_\mu t^{\mathbf{\Omega}_n}(\hat{v}_1, \dots, \hat{v}_n) = t^{\mathbf{2}}(\mu(v_1), \dots, \mu(v_n)), \tag{4.2.12}$$

hence $t = \sum_{\pi_\mu d = 1} v_1^{\mu_1} \cdots v_n^{\mu_n}$, so d codes the full DNF of t. □

We describe shortly the parallel algorithm for computing $d = t^{\mathbf{\Omega}_n}(b_1, \dots, b_n)$. Suppose we have a 2^k-bit processor at our disposal, $k < n$. Each vector b_i is divided into 2^{n-k} consecutive sequences. Hence, b_i consists of 2^{n-k} blocks b_{ij}, each of size 2^k. To find d, blocks $d_j = t(b_{1j}, \dots, b_{nj})$ of size 2^k are computed bitwise for $j = 1, 2, \dots, 2^{n-k}$. Then the combined vector $d_1 d_2 \dots d_{2^{n-k}}$ is the output vector d. The total time for computing d approximately is $T = 2^{l+n-k}\delta$, where 2^l is the total number of nodes in the binary expression tree of the term t and δ is the time interval for computing bitwise one logical operation[4].

Suppose now that we have 2^r 2^k-bit processors. Computations of d_j is distributed among them and they compute $t(b_{1j}, \dots, b_{nj})$ in parallel. Actually, they are acting as a single 2^{k+r}-bit processor. Hence, the total time for computing d in this case is $T = 2^{l+n-k-r}\delta$.

4.3 Computing finite models

Using a translation from $L_{\omega\omega}$ to L_ω, we are able to state and computationally solve various problems on finite structures.

[4]For modern computers, $\delta \approx 10^{-9}$ seconds

4.3.1 Translation from $L_{\omega\omega}$ to L_ω

A method for coding some notions, mostly of the combinatorial nature and related to countable first-order structures, by theories of propositional calculus L_{ω_1} is presented in [33]. The primary goal there was to study the complexity of these notions in Borel hierarchy. The coding is given there by a map $*$. We adapted this map for our needs.

Let L be a finite first-order language and $L_A = L \cup \{\underline{a} \mid a \in A\}$, where A is a finite nonempty set. Here $\underline{a}$ is a new constant symbol, the name of the element a. We define the set $\mathcal{P}$ of propositional letters as follows

$$\begin{aligned} \mathcal{P} = \ & \{p_{Fa_1\ldots a_k b} \mid a_1, \ldots, a_k, b \in A, F \text{ is a } k\text{-ary function symbol of } L\} \cup \\ & \{q_{Ra_1\ldots a_k} \mid a_1, \ldots, a_k \in A, R \text{ is a } k\text{-ary relation symbol of } L\} \end{aligned} \tag{4.3.1}$$

Then we define recursively a translation map $*$ from the set $Sent_{L_A}$ of all $L_{\omega\omega}$-sentences of L_A into the set of propositional formulas of $L_\omega^{\mathcal{P}}$. Basically, the map $*$ translates universal and existential quantifiers respectively into finite conjunctions and disjunctions of parameterized quantifier-free formulas.

$$\begin{aligned} &(F(\underline{a}_1, \ldots, \underline{a}_k) = \underline{b})^* \equiv p_{Fa_1\ldots a_k b}, \quad (R(a_1, \ldots, a_k))^* \equiv q_{Ra_1\ldots a_k}, \\ &(F(\underline{a}_1, \ldots, \underline{a}_k) = F'(\underline{a}'_1, \ldots, \underline{a}'_k))^* \equiv \\ &\quad \textstyle\bigwedge_{b\in A} (F(\underline{a}_1, \ldots, \underline{a}_k) = \underline{b})^* \Rightarrow (F'(\underline{a}'_1, \ldots, \underline{a}'_k) = \underline{b})^*), \\ &(F(t_1(\underline{a}_{11}, \ldots, \underline{a}_{1m}), \ldots, t_k(\underline{a}_{k1}, \ldots, \underline{a}_{km})) = \underline{b})^* \equiv \\ &\quad \textstyle\bigwedge_{(b_1,\ldots,b_k)\in A^k} \left(\bigwedge_{i=1}^k (t_i(\underline{a}_{i1}, \ldots, \underline{a}_{im}) = \underline{b}_i)^* \Rightarrow p_{Fb_1\ldots b_k b}\right), \\ &(R(t_1(\underline{a}_{11}, \ldots, \underline{a}_{1m}), \ldots, t_k(\underline{a}_{k1}, \ldots, \underline{a}_{km})))^* \equiv \\ &\quad \textstyle\bigwedge_{(b_1,\ldots,b_k)\in A^k} \left(\bigwedge_{i=1}^k (t_i(\underline{a}_{i1}, \ldots, \underline{a}_{im}) = \underline{b}_i)^* \Rightarrow q_{Rb_1\ldots b_k b}\right), \\ &(\neg\varphi)^* \equiv \neg\varphi^*, \quad (\varphi \wedge \psi)^* \equiv \varphi^* \wedge \psi^*, \quad (\varphi \vee \psi)^* \equiv \varphi^* \vee \psi^*, \\ &(\forall x \varphi(x))^* \equiv \textstyle\bigwedge_{a\in A} \varphi(\underline{a})^*, \quad (\exists x \varphi(x))^* \equiv \bigvee_{a\in A} \varphi(\underline{a})^*. \end{aligned} \tag{4.3.2}$$

If L has only one function symbol F, then we shall write $p_{a_1\ldots a_k b}$ instead of $p_{Fa_1\ldots a_k b}$. The similar convention is assumed for a relation symbol R. For example, if φ is the sentence which states the associativity of the binary function symbol $\cdot$, it appears that the $*$-transform of $i \cdot j = u$ is p_{iju} and that φ^* over domain $I_n = \{0, 1, \ldots, n-1\}$ is equivalent to

$$\bigwedge_{i,j,k,u,v,l<n} ((p_{iju} \wedge p_{jkv} \wedge p_{ukl}) \Rightarrow p_{ivl}) \tag{4.3.3}$$

We assume that the domain of a finite model $\mathbf{A}$ having n elements is $I_n = \{0, 1, \ldots, n-1\}$. If $\mathbf{A}$ is a model of L, note that the simple expansion $(\mathbf{A}, a)_{a\in A}$ is a model of L_A.

4.3.2 Correspondence between models of T and T^*

Using translation $*$, we give a method for constructing and counting finite models of first order theories for a finite language L. In this, the notion of a labeled model will have the important role. Therefore we fix this and related concepts.

Let $\mathbf{A}$ be a finite model of L, $|A| = n$. Any one-to-one and onto map $\alpha\colon I_n \to A$ will be called the labeling of $\mathbf{A}$. We can transfer the structure of $\mathbf{A}$ to a model $\mathbf{A}_\alpha$ with the domain I_n in the usual way:

1. If $R \in L$ is a k-placed relation symbol then we take
   $R^{\mathbf{A}_\alpha}(i_1, \ldots, i_k)$ iff $R^{\mathbf{A}}(\alpha(i_1), \ldots, \alpha(i_k)), i_1 \ldots, i_k \in I_n$.
2. If $F \in L$ is a k-placed function symbol then we take
   $F^{\mathbf{A}_\alpha}(i_1, \ldots, i_k) = \alpha^{-1}(F(\alpha(i_1), \ldots, \alpha(i_k)), i_1 \ldots, i_k \in I_n$.
3. If $c \in L$ is a constant symbol then $c^{\mathbf{A}_\alpha} = \alpha^{-1}(c^A)$.

We see that $\alpha\colon \mathbf{A}_\alpha \cong \mathbf{A}$. We shall call $\mathbf{A}_\alpha$ a labeled model of $\mathbf{A}$. Let $c_0, \ldots, c_{n-1}$ be new constant symbols to L and $L' = L \cup \{c_0, \ldots, c_{n-1}\}$. The simple expansion $(\mathbf{A}, \alpha_0, \ldots, \alpha_{n-1})$ is a model of L' such that c_i is interpreted by $\alpha_i = \alpha(i)$, $0 \leqslant i < n$. Instead of $(\mathbf{A}, \alpha_0, \ldots, \alpha_{n-1})$ we shall write shortly $(\mathbf{A}, \alpha)$.

Theorem 4.3.1. *Assume $\mathbf{A}$ is a finite model of L, $|A| = n$ and α, β are labelings of $\mathbf{A}$. Then the following are equivalent*

(i) $(\mathbf{A}, \alpha) \equiv (\mathbf{A}, \beta)$, *i.e.,* $(\mathbf{A}, \alpha)$ *and* $(\mathbf{A}, \beta)$ *are elementary equivalent models,*

(ii) $(\mathbf{A}, \alpha) \cong (\mathbf{A}, \beta)$,

(iii) $\mathbf{A}_\alpha = \mathbf{A}_\beta$,

(iv) $\alpha \circ \beta^{-1} \in \mathrm{Aut}(\mathbf{A})$.

Proof. It is well known that finite elementary equivalent models are isomorphic. Hence (i) is equivalent to (ii).

Suppose $(\mathbf{A}, \alpha_0, \ldots, \alpha_{n-1}) \cong (\mathbf{A}, \beta_0, \ldots, \beta_{n-1})$. So there is $f \in \mathrm{Aut}(\mathbf{A})$ such that $f(\beta_i) = \alpha_i$, $0 \leqslant i < n$. Hence $f \circ \beta = \alpha$, so $\alpha \circ \beta^{-1} \in \mathrm{Aut}(\mathbf{A})$. Therefore (ii) implies (iv). Reversing this proof, it also follows that (iv) implies (ii).

Suppose $(\mathbf{A}, \alpha) \equiv (\mathbf{A}, \beta)$ and let $F \in L$ be a k-placed function symbol. Then for any choice of constant symbols $c_{i_1}, \ldots, c_{i_{k+1}}$, $(\mathbf{A}, \alpha) \models F(c_{i_1}, \ldots, c_{i_k}) = c_{i_{k+1}}$ if and only if $(\mathbf{A}, \beta) \models F(c_{i_1}, \ldots, c_{i_k}) = c_{i_{k+1}}$. Hence

$$F^{\mathbf{A}}(\alpha(i_1), \ldots, \alpha(i_k)) = \alpha(i_{k+1}) \quad \text{iff} \quad F^{\mathbf{A}}(\beta(i_1), \ldots, \beta(i_k)) = \beta(i_{k+1}), \tag{4.3.4}$$

therefore $\alpha^{-1}(F(\alpha(i_1),\dots,\alpha(i_k)) = \beta^{-1}(F(\beta(i_1),\dots,\beta(i_k))$ for all $i_1,\dots,i_k \in I_n$. Thus we proved that $F^{(\mathbf{A},\alpha)} = F^{(\mathbf{A},\beta)}$. Similarly we can prove that $R^{(\mathbf{A},\alpha)} = R^{(\mathbf{A},\beta)}$ for each relation symbol $R \in L$. Hence we proved that $\mathbf{A}_\alpha = \mathbf{A}_\beta$ and so (i) implies (iii). Similarly one can prove that (iii) implies (i). □

Finite models of a first order theory T which have for domains the sets I_n are called labeled models of T. By $\mathscr{L}_{T,n}$ we shall denote the set of all labeled models of T of size n. By T_n we denote the theory $T \cup \{\sigma_n\}$, where σ_n denotes the sentence "there are exactly n elements". Therefore, $\mathscr{L}_{T,n}$ is the set of all labeled models of T_n.

By a finite theory we mean a first order theory T with finitely many axioms, i.e., T is a finite set of sentences of a finite language L. We can replace T with a single sentence, but in some cases we need to add or remove a sentence from T. In these cases, it is technically easier to work with a set of sentences then with a single sentence which replaces T.

Suppose T is a finite theory. Let $\mathcal{P}$ be the set of propositional letters defined by (4.3.1) over $A = I_n$ and the language L, and let $T^* = \{\varphi^* \mid \varphi \in T\}$. Further, let $\mathfrak{M}(T^*) \subseteq 2^{\mathcal{P}}$ denote the set of all models of T^*, i.e., valuations satisfying all propositional formulas in T^*. The next construction describes the correspondence between labeled models of T and models of T^*.

The function h which assigns to each $\mu \in \mathfrak{M}(T^*)$ a labeled model $h(\mu) = \mathbf{A}$ of T is defined as follows. Let $a_1,\dots,a_k,b \in I_n$. Then

If $F \in L$ is a k-placed function symbol, then

$$F^{\mathbf{A}}(a_1,\dots,a_k) = b \quad \text{iff} \quad \mu(p_{Fa_1\dots a_k b}) = 1. \tag{4.3.5}$$

If $R \in L$ is a k-placed relation symbol, then

$$\mathbf{A} \models R[a_1,\dots,a_k] \quad \text{iff} \quad \mu(q_{Ra_1\dots a_k}) = 1. \tag{4.3.6}$$

By induction on the complexity of the formula φ, we proved:

Theorem 4.3.2. *The map h codes the models in $\mathscr{L}_{T,n}$ by models of T^*.*

This theorem is our starting point in determining finite models of T of size n. Since T^* is finite, we can replace it with a single propositional formula $\theta = \bigwedge_{\psi\in T^*}\psi$. Obviously, we may consider θ as a Boolean term $t(v_1,\dots,v_m)$. Computing $t^{\mathbf{\Omega}_m}(b_1,\dots,b_m)$ in free Boolean algebra $\mathbf{\Omega}_m$ for free generators $b_1,\dots,b_m$, we obtain the vector b which by Proposition 4.2.2 codes the full DNF of θ, hence all models of T^*. This gives us all labeled models of T of size n via the map h.

Let $l_{T,n}$ denote the cardinality of $\mathscr{L}_{T,n}$. Obviously, $l_{T,n}$ is equal to the number of bits in the vector b that are equal to 1.

The major target in finite model theory is to count or to determine non-isomorphic models of T of size n. By $\mathfrak{M}(T)_n$ we denote a maximal set of non-isomorphic models of T with the domain I_n. Elements of this set are also called unlabeled models of T. By $\kappa_{T,n} = |\mathfrak{M}(T)_n|$ we denote the number of non-isomorphic (unlabeled) models of T of size n. If a theory T is fixed in our discussion, we often omit the subscript T in these symbols. In other words, we shall simply write $\mathscr{L}_n$, l_n, $\mathfrak{M}_n$ and κ_n. In our examples, the following theorem will be useful in finding numbers l_n and κ_n.

Theorem 4.3.3. (*Frobenius - Burnside counting lemma*) *Let* $\mathbf{A}$ *be a finite model,* $|A| = n$. *Then the number of models isomorphic to* $\mathbf{A}$ *which have the same domain* A *is equal to* $n!/|\mathrm{Aut}(\mathbf{A})|$.

If T *is a theory of a finite language* L *with finite number of axioms, then*

$$l_n = \sum_{\mathbf{A} \in \mathfrak{M}_n} \frac{n!}{|\mathrm{Aut}(\mathbf{A})|}. \tag{4.3.7}$$

Note that this theorem immediately follows from Theorem 4.3.1 and direct application of Langrange's subgroup theorem on the symmetric group S_n of I_n.

It is said that a set of models $\mathcal{K}$ is adequate for n-models of T if $\mathfrak{M}_n \subseteq \mathcal{K} \subseteq \mathscr{L}_n$. Even for small n the set $\mathscr{L}_n$ can be very large. On the other hand, it is possible in some cases to generate easily all labeled models, or to determine l_n from $|\mathcal{K}|$ for an adequate family $\mathcal{K}$ of the reasonable size. Also, it is commonly hard to generate directly non-isomorphic models of T, or to compute κ_n. But for a well chosen adequate set of models these tasks can be done. Adequate families are usually generated by filtering $\mathscr{L}_n$, fixing some constants or definable subsets in models of T, or imposing extra properties, for example by adding new sentences to T. In our examples we will give some instances of adequate families.

4.4 Software implementation

When we started the development of the system, our main motivation was the algorithm described in §4.2.3. So it should come as no surprise that the parallel model counting is its main feature. However, that is just one aspect of the overall system. In addition, we also developed features such as: a DSL for stating first-order theories, their translation into propositional calculus, and partial evaluation of propositional formulas to name a few prominent ones.

The entire system is implemented in Python programming language, with the exception of hardware accelerated primitives such as the one used for parallel model counting. They are implemented using OpenCL, which is a framework for parallel computing. But first, before we get into the details, let us start with some high-level implementation considerations.

Ideally, the system would be used in the following way:

(i) We specify a first-order theory that we are interested in.

(ii) The system finds the number of models of the given theory.

And while this out-of-the-box approach works for small examples, that is, for first-order theories on small domains, in principle that will not be the case. The reason is that the general model counting algorithms do not scale well with an increase in the domain size. On the contrary, a model counting algorithm that is tailored for the first-order theory at hand can scale much better. However, implementing such an algorithm is non-trivial to say the least.

That is why we devised the system as a sort of a toolkit for creating problem specific model counters, or in other words, as a Python library that provides all the necessary building blocks for their construction. The idea is to allow for custom model counters to be developed in the same way as any other Python program. Hence, the important goal of the implementation was to ensure that the functionality provided by the system is well integrated with Python.

In the rest of the section we will go through the implementation details of each important feature of the system. Specifically, in the first subsection we will explain how first-order theories are specified and how they are translated into the propositional form. Also, we will show how propositional formulas are represented within the system.

In the second subsection we will introduce the built-in support for the development of propositional formula manipulations. Most importantly, we will show how the partial evaluation of propositional formulas works.

In the third subsection we will go over the inner-workings of hardware acceleration, and in the fourth subsection we will present hardware accelerated primitives that can be used to develop custom model counters.

Finally, in the fifth subsection we will give an example that demonstrates how different features of the system fit together.

4.4.1 Representation of first-order theories

Propositional formulas occupy a central place within the system. There are several reasons for this. The most important one is that the OpenCL C code for hardware acceleration is generated on the basis of propositional formulas. Therefore, we needed a data structure for representing propositional formulas that supports a straightforward code generation. One such data structure is called an abstract syntax tree or AST.

ASTs are commonly used in compilers to represent the syntactic structure of the source programs [2]. They are considered abstract in the sense that they do not represent every detail

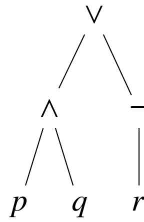


Fig. 4.1 AST for $p \wedge q \vee \neg r$

appearing in the real syntax, such as inessential punctuation and delimiters. As a result, ASTs are also very suitable for programmatic manipulation and processing.

Example 4.4.1. Let us consider the following propositional formula

$$p \wedge q \vee \neg r. \tag{4.4.1}$$

As can be seen in Fig. 4.1, the appearance of AST is fairly uniform. That is, each interior node represents an operator, and consequently the children of the node represent the operands of the operator. This also means that the operator precedence rules and the grouping of operands are naturally embodied within the structure of an AST. In other words, ASTs are unambiguous, i.e., if we reconstruct the formula from the AST we will get

$$((p \wedge q) \vee (\neg r)),$$

which is exactly the fully parenthesized version of (4.4.1).

The formal language of propositional logic (in the sense of §2.2) that we are interested in is $L = \{\wedge, \vee, \veebar, \neg, \top, \bot\}$, where $\mathrm{Fnc}^2_L = \{\wedge, \vee, \veebar\}$ are binary operation symbols, $\mathrm{Fnc}^1_L = \{\neg\}$ is an unary operation symbol, and $\mathrm{Const}_L = \{\top, \bot\}$ are constant symbols. Based on §2.3, we can say that each function symbol $F \in \mathrm{Fnc}^k_L$ has associated F-terms, that is, terms $F(t_1, \ldots, t_k)$, where $t_1, \ldots, t_k \in \mathrm{Term}_L$. So, in order to build ASTs of terms of the language L, we need a distinct AST node type corresponding to each function symbol F, as well as one for constants and one for variables.

Each node type is implemented as a separate subclass of an abstract `Term` class that represents terms of the language. Hence we have the following Python classes:

`class` **`Not`**`(t)`

Represents ¬-terms of the language.

Each ¬-term represents the result of application of operation denoted by ¬ symbol to its input term `t`.

`class And(t1, t2)`

Represents ∧-terms of the language.

Each ∧-term represents the result of application of operation denoted by ∧ symbol to its input terms `t1` and `t2`.

`class Or(t1, t2)`

Represents ∨-terms of the language.

Each ∨-term represents the result of application of operation denoted by ∨ symbol to its input terms `t1` and `t2`.

`class Xor(t1, t2)`

Represents $\veebar$-terms of the language.

Each $\veebar$-term represents the result of application of operation denoted by $\veebar$ symbol to its input terms `t1` and `t2`.

`class Var(name, *indices)`

Represents variables.

Each variable has a name and zero or more indices. For example, `Var('p')` can be used to represent p, and similarly `Var('p', 1, 2)` for p_{12}.

`__call__(self, *indices)`

Shorthand for creating new indexed variables out of the existing one, which in case `p` is `Var('p')`, or formally in Python `p = Var('p')`, allows the use of `p(1, 2)` instead of `Var('p', 1, 2)`.

`class Const(value)`

Represents constants of the language.

Each constant has a value, i.e., the interpretation.

`false = Const(0)`

Represents ⊥ constant of the language.

`true = Const(-1)`

Represents ⊤ constant of the language.

We used -1 in the definition of `true` because Python has arbitrary precision integers represented in 2's complement. Hence, -1 is an integer with all bits set, i.e., it corresponds to a bit vector with all ones.

Using the introduced classes, it is possible to create an AST of any term of the language L. For example, the AST of the formula (4.4.1) can be created by the following Python expression

```
Or(And(Var('p'), Var('q')), Not(Var('r'))).
```

However, creating ASTs in such a way is quite tedious (especially for more complex formulas). Moreover, the resulting Python expressions are not easily understood, i.e., it is hard to say to which propositional formula a given Python expression corresponds. So, in order to simplify the input of formulas into the system, we developed a domain-specific language (DSL) for creating their ASTs.

DSLs are small computer languages of limited expressiveness, and unlike the general-purpose ones, they are focused on a particular kind of problem or domain. Their value stems from the fact that a well-designed DSL can be much easier to program with than a traditional software library. Hence, the obvious goal for the design of our DSL was to resemble the language of propositional logic as much as possible. In particular, we wanted to emulate the declarative style that is present when formulas are stated mathematically.

Propositional variables and connectives

The basic idea behind the implementation of our DSL is simple: we want to write propositional formulas as Python expressions and to get the corresponding ASTs as a result of their evaluation. Therefore, we implemented our DSL in Python as an internal DSL [14].

This design has the advantage of not requiring the development of a custom parser, and it integrates well with the rest of the system. For example, we can use AST nodes directly in Python expressions. However, Python clearly does not have operators that exactly correspond to logical connectives. Since our DSL syntax is limited by the Python syntax, we need to reuse some of the existing Python operators to implement logical connectives.

Considering the domain of our DSL, out of all Python operators, bitwise operators are especially suitable for our needs. Namely, they are semantically very close to the corresponding logical connectives, and, more importantly, they behave the same in terms of parsing. In particular, the grouping of operands and the precedence rules are the same.

Thanks to the fact that Python supports operator overloading, the implementation of logical connectives is straightforward. We simply customized the bitwise operators for the `Term` class to behave as shown in Table 4.1.

As a result, to create an AST of a propositional formula, it is sufficient to rewrite it as an equivalent Python bitwise expression over AST variables (`Var` nodes). In other words, the AST of the formula (4.4.1) is created in the following way

```
p & q | ~r,
```

where `p`, `q`, and `r` are `Var('p')`, `Var('q')`, and `Var('r')`, respectively.

Expression	Resulting AST
`~t`	`Not(t)`
`t1 & t2`	`And(t1, t2)`
`t1 | t2`	`Or(t1, t2)`
`t1 ^ t2`	`Xor(t1, t2)`

Table 4.1 Bitwise operations over ASTs

4.4.2 Partial evaluation of propositional formulas

When generating OpenCL C code, the number of variables of the generated function corresponds to the number of free variables of the initial propositional formula. Therefore, if some free variables are constant, the generated code will not be optimal. Conversely, if we know that some variables are constant or if we have intentionally fixed the values of some variables, we can use this knowledge to generate optimized OpenCL C code.

Optimization of functions through specialization when some of their inputs are fixed to constant values is known as partial evaluation. The basic theory and methods of partial evaluation in computer science are well explained in [22]. In our case, we want to specialize the AST of the initial propositional formula. That is, we want to get a new AST that corresponds to the initial propositional formula with some free variables replaced by their constant values. Moreover, we want to produce a reduced AST, without logical constants `true` and `false`, except when the AST itself is a constant.

AST manipulations are usually based on the Visitor design pattern [15], as it allows the addition of new functionality to the AST class hierarchy without requiring modification of the existing AST classes. In order to facilitate the development of new visitors, the system provides the following basis:

`class` **`Visitor`**

Base class for AST visitors.

It provides a default AST traversal, and dispatching to the appropriate handling method for a node based on the node's class.

`__call__``(self, node, *args, **kwargs)`

Shorthand for invoking the `visit()` method by calling the visitor object itself, thus allowing visitors to be used as properties.

visit(self, node, *args, **kwargs)
Dispatches the call to a method suitable for handling the given node.

The selection is done by searching the visitor object for methods with names of the form 'visit_' + cls.__name__, where cls is the class (or superclass) of the given node. If nothing is found, the default_visit method is selected.

default_visit(self, node, *args, **kwargs)
Continues the traversal of the AST by visiting the children of the given node.

New visitors are implemented by subclassing as seen in Listing 4.1, which shows the implementation of the partial evaluation visitor.

```
1  class TEval(Identity):
2      """Partially evaluates an AST."""
3
4      def visit_Var(self, var, values):
5          val = values.get(var)
6          if val in {true, 1, 'T', '1'}:
7              return true
8          elif val in {false, 0, 'F', '0'}:
9              return false
10         return var
```

Listing 4.1 Partial evaluation visitor

The TEval extends the Identity visitor (line 1), a utility visitor provided by the system, which is used to duplicate the AST during a visit. In order to have the variables replaced with the corresponding constants in the newly created AST, we customized the handling of Var nodes by implementing the visit_Var method (line 4). The method by itself is straightforward. It simply looks up the given values dictionary (line 5) to find out whether the given variable has a constant value and returns the appropriate constant if so (lines 7 and 9).

The partial evaluation visitor is not intended to be used directly. Instead, it is used as shown in Example 4.4.2, that is, its functionality is exposed as a method of the following class:

class **Expression**(ast)
Represents an AST that can be evaluated.

eval(values)
Returns a partially evaluated AST using the given values.

Example 4.4.2. Suppose that we already have an expression e that corresponds to (4.4.1), i.e., e = Expression(p & q | ~r). Now suppose that we know that q is always false. Then we could use partial evaluation to specialize e for known facts. Specifically, by executing

```
s = e.eval({q: false}),
```

we get a new expression `s` in which all occurrences of `q` are replaced by `false`. In other words, the expression `s` is as if it was created as follows

```
s = Expression(p & false | ~r).
```

If we take a closer look at the formula that corresponds to the specialized expression `s` from Example 4.4.2, it is obvious that it can be further simplified. Namely,

$$p \land \bot \lor \neg r \Leftrightarrow \neg r.$$

That is, we can eliminate another propositional variable: p. But to achieve this, it is necessary to use symbolic computations, which is, unsurprisingly, one of the techniques commonly used for partial evaluation [22].

To carry out symbolic computations at the AST level, we use the following facts:

(i) Our DSL is itself valid Python.

(ii) During object instantiation, Python allows for changing the type of the resulting object.

The idea is that while creating an AST, we perform symbolic computations by replacing nodes with their simplifications whenever possible. We accomplished this goal by customizing instantiation of the following AST classes:

`Not.__new__(cls, *terms, **kwargs)`

Creates a new `Not` node or its simplification.

Simplification is done if any of the following identities of propositional logic can be applied to the given `terms`:

$$\neg\top \Leftrightarrow \bot, \quad \neg\bot \Leftrightarrow \top, \quad \neg\neg x \Leftrightarrow x.$$

`And.__new__(cls, *terms, **kwargs)`

Creates a new `And` node or its simplification.

Simplification is done if any of the following identities of propositional logic can be applied to the given `terms`:

$$x \land \bot \Leftrightarrow \bot, \quad x \land \top \Leftrightarrow x, \quad x \land x \Leftrightarrow x, \quad x \land \neg x \Leftrightarrow \bot.$$

`Or.__new__(cls, *terms, **kwargs)`

Creates a new `Or` node or its simplification.

Simplification is done if any of the following identities of propositional logic can be applied to the given `terms`:

$$x \vee \top \Leftrightarrow \top, \quad x \vee \bot \Leftrightarrow x, \quad x \vee x \Leftrightarrow x, \quad x \vee \neg x \Leftrightarrow \top.$$

`Xor.__new__(cls, *terms, **kwargs)`

Creates a new `Xor` node or its simplification.

Simplification is done if any of the following identities of propositional logic can be applied to the given `terms`:

$$x \veebar \top \Leftrightarrow \neg x, \quad x \veebar \bot \Leftrightarrow x, \quad x \veebar x \Leftrightarrow \bot, \quad x \veebar \neg x \Leftrightarrow \top.$$

It is worth noting that the way in which symbolic computations are implemented works equally well for both DSL and partial evaluation. For example, as a result of symbolic computations, the following comparison is true

```
p & false | ~r == ~r,
```

and so is its equivalent expressed using partial evaluation

```
Expression(p & q | ~r).eval({q: false}) == Expression(~r).
```

4.4.3 Hardware acceleration

As already stated, the primary goal of this thesis is to implement model counting utilizing all available hardware resources. We have shown that this task is equivalent to computing an appropriate Boolean term t with n Boolean variables. More precisely, in §4.2 we have shown that this is equivalent to evaluating t in n free Boolean vectors of $\mathbf{\Omega}_n$.

In §4.2.3 we have outlined the parallel evaluation algorithm, where we suggested the use of bitwise instructions on all cores of available CPUs or GPUs. In the rest of this subsection we present the details of the OpenCL implementation of this algorithm.

The OpenCL framework models a computation as a multidimensional array of threads, where each thread executes the same compute kernel[5], but on different data. Not surprisingly, a compute device itself is modeled as an array of cores. So, in order to achieve parallelization, our task is to model the data with the suitable array, and the framework will do the heavy lifting.

[5] In the OpenCL framework, a compute kernel is a function declared in an OpenCL program and executed on a compute device such as a CPU or GPU.

In other words, we specify the shape of the data array, as well as the shape and number of its pieces, and let the framework allocate a thread for each piece of the data array and schedule it on an appropriate core for execution. During execution, the connection between threads and data is maintained by common indices of the corresponding thread and data arrays. This mechanism allows kernels to find out which piece of the data array they should work with.

Our data model already fits quite nicely with the OpenCL framework as we are working with segments of free Boolean vectors, i.e., segments of bit arrays. For example, let t be a propositional formula with n free variables, and let the size of segments that we can work with be 2^k. Now suppose that we want to evaluate t on a processor with 64-bit registers. This would mean that $k = 6$, and that we need to evaluate 2^{n-6} segments in order to evaluate the whole formula. Therefore, we have the following statement.

Proposition 4.4.1. *OpenCL data model for the evaluation of a propositional formula t with n free variables on a 2^k-bit processor is a one-dimensional array of the size 2^{n-k}.*

It should be noted that, as a consequence of the corollary, the computation in OpenCL is also modeled as a one-dimensional array of threads.

The size of the thread array depends on the size of the piece of the data array that each thread needs to process. In our case, this means that the number of threads depends on the number of segments of free Boolean vectors that are processed by each thread. We use a thread per segment, but other partitions are also possible. As a result, we have a 1-1 mapping between threads and segments. Or in other words, a compute kernel needs to implement the processing of a single segment.

The source code of compute kernels is generated from the corresponding templates by replacing placeholders (demarcated with single or double curly braces) with the appropriate content. There are two types of placeholders: hardware-dependent and formula-dependent. Therefore, as a first step, the template is adapted to the actual hardware by replacing only hardware-dependent placeholders. Then, in the second step, the source code adapted to the actual propositional formula is generated by replacing the remaining formula-dependent placeholders.

As we shall see in the next subsection, the system uses multiple compute kernels. One for each accelerated primitive it provides. However, all kernels share a common function `f` that implements the evaluation of a propositional formula t. Hence, this function represents the essence of our compute kernels.

Evaluation function

The template in Listing 4.2 contains an incomplete set of OpenCL C instructions. Our system uses this template and the AST of a propositional formula t to generate the OpenCL function `f` (line 1) that evaluates t.

```
1  {cl_type} f(uint gid, __local {cl_type} *c, __local {cl_type} *p) {{{{
2      bool v[{{1}}];
3      uint m = 1;
4      for (uint j=0; j < {{1}}; j++) {{{{
5          v[j] = gid & m;
6          m <<= 1;
7      }}}}
8
9      return {{2}};
10 }}}}
```

Listing 4.2 Evaluation function template

Specifically, `{cl_type}` (line 1) is replaced with the proper OpenCL data type that best suites the size of the actual registers. For example, on some hardware that could be a scalar data type like 64-bit `long`, while on other a vector[6] data type like `int4` would be more appropriate. Similarly, `{{1}}` (lines 2 and 4) and `{{2}}` (line 9) are replaced with the values that are specific to the formula t. In particular, `{{1}}` is a value derived from the number of variables, while `{{2}}` is a bitwise expression in OpenCL C that is generated from the AST of t.

In order to explain the implementation of `f`, we need a context that is best illustrated by the following example. Suppose that we have a 4-bit processor and that we want to evaluate a formula with five propositional variables. According to Proposition 4.4.1, the partitioning of free Boolean vectors into segments is shown in Table 4.2.

The implementation of `f` is based on the observation that the segments of some vectors are constant, and that the segments of the remaining vectors can take only two values. In our example, segments of ω_0, ω_1 and ω_2 take only two values, either $a_0 = 0000$ or $a_1 = 1111$, while segments of ω_3 and ω_4 have constant values $b_0 = 0011$ and $b_1 = 0101$. Observe that a_0 and a_1 are $\mathbf{0}$ and $\mathbf{1}$ vectors, and that b_0 and b_1 are free vectors of $\mathbf{\Omega}_2$. In general, this arrangement property of free Boolean vectors $\omega_0, \ldots, \omega_{n-1}$ follows from the next isomorphism of algebraic structures:

$$\mathbf{\Omega}_n \simeq \mathbf{\Omega}_k^{2^{n-k}}.$$

[6] The OpenCL vector data types are composed of scalar components, thus `int4` has four 32-bit `int` elements, which effectively makes it 128-bit.

Free vectors	Segments							
	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7
ω_0	0000	0000	0000	0000	1111	1111	1111	1111
ω_1	0000	0000	1111	1111	0000	0000	1111	1111
ω_2	0000	1111	0000	1111	0000	1111	0000	1111
ω_3	0011	0011	0011	0011	0011	0011	0011	0011
ω_4	0101	0101	0101	0101	0101	0101	0101	0101

Table 4.2 A partitioning of free Boolean vectors of $\mathbf{\Omega}_5$

Therefore, `f` is invoked with the following arguments:

(i) `gid`, a segment index.

(ii) `c`, an array of free vectors of $\mathbf{\Omega}_k$.

(iii) `p`, an array containing the vectors **0** and **1** of $\mathbf{\Omega}_k$.

The evaluation of t for the given `gid` is performed using the bitwise expression generated from the AST of t. The segments of free vectors of $\mathbf{\Omega}_n$ that correspond to `gid` are looked up using arrays `c`, `p`, and `v`, where `v` contains the binary expansion of `gid`. The generated expression uses these segments to compute the corresponding segment of the resulting vector, which is also the return value of `f`. In our example, the segments of ω_0, ω_1, ω_2, ω_3, and ω_4 are contained in `p[v[0]]`, `p[v[1]]`, `p[v[2]]`, `c[0]`, and `c[1]`, respectively.

4.4.4 Accelerated primitives

The system provides a number of hardware accelerated Python primitives that can be used when developing custom model counters. They are exposed as methods of the following class:

`class ClRunner(device)`

Represents an execution engine that utilizes hardware acceleration to compute the results of provided primitives.

Each runner (an instance of `ClRunner`) has a `device`, the actual OpenCL compute device used for execution.

`eval(expression)`

Computes all models of the given `expression` using parallel evaluation.

`count(expression)`

Computes the number of models of the given `expression` using parallel counting.

findmodel(expression)

Computes a model of the given `expression` using parallel search.

exist(expression)

Returns `True` if the given `expression` has a model, `False` otherwise.

As a convenience, the system provides the following predefined runners:

class **GPURunner**

A `ClRunner` subclass that uses the GPU with the largest number of cores for execution.

class **CPURunner**

A `ClRunner` subclass that uses the CPU for execution.

For example, to find the number of models of the formula (4.4.1) using a GPU, we would execute the following

```
GPURunner().count(Expression(p & q | ~r)).
```

We note that although CPUs do not have as many cores as GPUs, the difference in performance is not proportional to the difference in the core count. In fact, the performance of CPUs is better than one could conclude from the core count alone. The reason is that CPUs usually have three times higher operating frequency than GPUs. Moreover, the OpenCL framework is actually using advanced CPU instructions such as SSE or AVX, which in turn utilize wider registers (128-bit or even 256-bit). Therefore, high-end CPUs are more like low-to-mid-range GPUs in terms of performance of our algorithms.

In the rest of this subsection we will explain the implementation of the provided primitives in more details.

eval

Suppose we are given a Boolean expression t with n variables. We remind that

$$t^{\mathbf{\Omega}_n}(\omega_1, \ldots, \omega_n) = \omega \tag{4.4.2}$$

represents the value of the expression t in free Boolean algebra $\mathbf{\Omega}_n$, where $\omega_1, \ldots, \omega_n$ are free Boolean vectors.

When evaluating expression t, the number of variables m is constrained by the size of the available memory M (in bytes) of a compute device that is used for evaluation. Namely, m is the largest integer such that 2^m bits is less than or equal to $8M$, i.e.,

$$m = \lfloor \log_2(8M) \rfloor. \tag{4.4.3}$$

This size limit is a natural consequence of the need to store the resulting vector ω for further processing. Therefore, the `eval` primitive implements parallel evaluation of Boolean expressions with at most m variables.

The computation itself is implemented in OpenCL using a dedicated kernel. However, OpenCL kernels cannot be invoked directly from Python. Hence, the major part of the implementation of a primitive deals with the preparations for executing the corresponding kernel. These preparations are specific to the OpenCL framework. Basically the implementation of a primitive must perform the following:

(i) Prepare an OpenCL context and a command queue for interacting with the compute device.

(ii) Generate the source code of an OpenCL program using the template of the corresponding kernel and the given expression t.

(iii) Build (compile & link) an executable from the program source.

(iv) Allocate a memory buffer on the compute device for storing the resulting vector ω.

(v) Enqueue the corresponding kernel for execution on the compute device.

(vi) Retrieve the result from the compute device.

The template of the `eval` kernel is shown in Listing 4.3. The kernel itself is essentially a thin wrapper around the evaluation function `f`, and accordingly its main task is to prepare the arguments necessary for calling `f`. As explained in §4.4.3, `f` is called with the appropriate constants from $\mathbf{\Omega}_k$, where k is determined by the size of registers of the used compute device (see Proposition 4.4.1).

Therefore, during hardware specialization of the template, the declaration of array `c` is completed by replacing `{const_vars}` (line 3) with k as this is the number of free vectors in $\mathbf{\Omega}_k$. The initialization of array `p` is completed by replacing `{cl_false}` (line 5) and `{cl_true}` (line 6) with $\mathbf{0}$ and $\mathbf{1}$ vectors of $\mathbf{\Omega}_k$. Finally, `c` is initialized to contain free vectors of $\mathbf{\Omega}_k$ by replacing `{const_init}` (line 8) with appropriate instructions. As before, `{cl_type}` is replaced with the OpenCL data type that is most suitable for the used compute device.

The rest of the kernel is straightforward. It simply retrieves the index of the segment[7] to be evaluated (line 10), and calls the evaluation function `f` to do the actual work (line 11). The computed value is stored in the memory buffer `result`, allocated for the resulting vector ω.

[7] Technically, `get_global_id` returns the thread index, but due to 1-1 mapping between threads and segments, it is also a segment index.

```
1    __kernel void eval(__global {cl_type} *result) {{{{
2        __local {cl_type} p[2];
3        __local {cl_type} c[{const_vars}];
4
5        p[0] = {cl_false};
6        p[1] = {cl_true};
7
8        {const_init}
9
10       uint gid = get_global_id(0);
11       result[gid] = f(gid, c, p);
12   }}}}
```

Listing 4.3 Template of the `eval` kernel

`count`

Suppose t is a Boolean expression with n variables. In order to compute $\kappa(t)$, the number of models of t, the `count` primitive implements parallel counting of models in the vector ω, which is the result of evaluating t in free Boolean algebra $\mathbf{\Omega}_n$ as in (4.4.2). We remind that

$$\omega = (\omega(1), \omega(2), \dots, \omega(2^n))$$

is a bit vector of the size 2^n. As explained in Proposition 4.2.2, and according to (4.2.12), if $\omega(k) = 1$, then $(\omega_1(k), \omega_2(k), \dots, \omega_{2^n}(k))$ is the corresponding model of t. Hence, the computation of $\kappa(t)$ is reduced to counting the number of bits in ω that are set to 1.

In general, the implementation of the `count` primitive closely resembles the implementation of the `eval` primitive. The biggest difference is that, unlike the `eval` primitive, the `count` primitive does not limit the number of variables in Boolean expressions. Therefore, in case that n is greater than m defined by (4.4.3), it performs a partitioning of the problem by reducing t across all valuations μ from $\mathbf{2}^{n-m}$. Reduction is performed in Python by partially evaluating t (as explained in §4.4.2) under the assumption that μ is assigned to the first $n - m$ variables of t. This effectively replaces the starting problem over $\mathbf{\Omega}_n$ with 2^{n-m} instances over $\mathbf{\Omega}_m$. Let t_μ denote the reduction of t corresponding to the valuation $\mu \in 2^{n-m}$. Then, the total number of models of t is

$$\kappa(t) = \sum_{\mu \in 2^{n-m}} \kappa(t_\mu). \tag{4.4.4}$$

While we could have implemented the `count` primitive by evaluating the reduced expressions t_μ using the `eval` primitive and processing the resulting vectors ω_μ in Python, we did not. Instead, we use the `count` kernel given in Listing 4.4 to perform the entire counting on the

compute device itself. This completely eliminates the transfer and processing of vectors ω_μ, thus improving performance.

```
1   __kernel void count(__global {cl_type} *result) {{{{
2       __local {cl_type} p[2];
3       __local {cl_type} c[{const_vars}];
4
5       p[0] = {cl_false};
6       p[1] = {cl_true};
7
8       {const_init}
9
10      uint gid = get_global_id(0);
11      result[gid] += ({cl_type}) ({{3}}) * popcount(f(gid, c, p));
12  }}}}
```

Listing 4.4 Template of the `count` kernel

`findmodel`

Suppose t is a Boolean expression with n variables. In order to find a model of t, the `findmodel` primitive implements parallel search for a model in the vector ω, which is the result of evaluating t in free Boolean algebra $\mathbf{\Omega}_n$ as in (4.4.2). We remind that ω is a bit vector, and that each bit set to 1 corresponds to a model of t as explained in Proposition 4.2.2. Therefore, the search for a model of t is reduced to finding a bit in ω that is set to 1.

As with the `count` primitive, the `findmodel` primitive also does not limit the number of variables in Boolean expressions. In fact, their implementations are essentially equivalent, and the only difference is in the used kernel. The `findmodel` primitive uses the `findmodel` kernel shown in Listing 4.5.

The evaluation of the expression t is performed using the evaluation function `f`, hence for the most part the `findmodel` kernel is the same as the `eval` kernel in Listing 4.3. What is specific to the `findmodel` kernel is `{cl_condition}` (line 12), which is replaced during hardware specialization of the template by the condition appropriate for the actual `{cl_type}`. Depending on whether it is a scalar or vector OpenCL data type, the possible conditions are:

- `popcount(f(gid, c, p))` for scalar data types, and
- `any(popcount(f(gid, c, p)) != 0)` for vector data types.

However, in both cases the condition works the same, i.e., it checks whether the computed segment contains any models.

```
__kernel void findmodel(__global uint *result) {{{{
    __local {cl_type} p[2];
    __local {cl_type} c[{const_vars}];

    p[0] = {cl_false};
    p[1] = {cl_true};

    {const_init}

    if (!result[0]) {{{{
        uint gid = get_global_id(0);
        if ({cl_condition}) {{{{
            result[0] = gid;
        }}}}
    }}}}
}}}}
```

Listing 4.5 Template of the `findmodel` kernel

In other words, the `findmodel` kernel keeps computing segments of the resulting vector ω until it finds a segment that contains a model. Once the segment is found, its index is returned in the memory buffer `result` (line 13), and the actual model is then reconstructed in Python.

As an interesting remark, notice that the kernel itself is not thread-safe (line 10). We used the fact that all models are adequate to avoid the use of synchronization. As a result, we do not know which index will be returned in case that multiple threads find a segment with a model at the same time.

`exist`

The `exist` primitive is a convenience primitive that accomplishes its task by delegating the work to the `findmodel` primitive. It returns `True` if the `findmodel` primitive finds a model, `False` otherwise.

4.4.5 Example

Classic 9x9 Sudoku is a well-known problem, not hard to comprehend, and yet it requires a custom solver for solving it with our system. Therefore, it is a convenient example to show how various features of the system fit together, and how overall integration with Python works.

The Sudoku problem is stated as follows. Suppose we have a partial quasigroup $\mathbf{Q} = (Q, *)$, $Q = \{1, 2, \ldots, 9\}$, that is, we have $i * j = k$ for certain triplets $(i, j, k) \in Q^3$, which satisfy the

following axioms

$$i * j = i * j' \Rightarrow j = j', \quad i * j = i' * j \Rightarrow i = i'.$$

In addition, **Q** is a special quasigroup, i.e., it has the property that when its table is divided into nine 3x3 squares, each square contains all the elements of Q. So, the Sudoku problem is to complete the table of **Q** while respecting the stated rules.

In our approach the quasigroup operation $*$ is represented by ternary relation $p(i, j, k)$, meaning $i * j = k$. In our system $p(i, j, k)$ is read as a propositional variable p with 3 indices i, j, and k. Therefore the statement of the Sudoku problem is as given in the Listing 4.6.

```
1  # there is a value in every cell
2  f1 = A[i : S].A[j : S].E[k : V](p(i, j, k))
3
4  # each value appears in each row
5  f2 = A[i : S].A[k : V].E[j : S](p(i, j, k))
6
7  # each value appears in each column
8  f3 = A[j : S].A[k : V].E[i : S](p(i, j, k))
9
10 # in one cell there can be only one value
11 f4 = A[i : S].A[j : S].A[k, l : Dkl](~(p(i, j, k) & p(i, j, l)))
```

Listing 4.6 Latin square properties

The specification is straightforward, we state that there is a value in every cell (line 2), that each value appears in each row (line 5) and each column (line 8), and that there cannot be multiple values in any cell (line 11). This declarative style of specification was the primary motive for the development of our DSL for first-order logic.

To complete the Sudoku specification, we need an additional property that is given in Listing 4.7. It states that each value appears in each subsquare.

```
1  # each value appears in each subsquare
2  f5 = bool.true
3  for sx, sy in prod(range(0, n**2, n), repeat=2):
4      f5 &= A[k : V].E[i, j : subsquare(sx, sy)](p(i, j, k))
```

Listing 4.7 Sudoku property

As can be seen we use the integration with Python to construct the propositional formula `f5` (line 2) in an imperative manner. We loop through all subsquares (line 3) and build the final conjunction by adding formulas for each subsquare one by one (line 4).

The complete source of the program is given in §A.1.

4.5 Comparison against similar software

There are several software systems that are used for model generating and counting. They range from general finite model finders like Mace4 [27] and Paradox [8], to more specialized ones like PSATO [42] which was developed for computing specific quasigroups. Also, it is worth noting that there is a rich literature on applications of these systems and algorithms in solving problems in finite combinatorics, models, as well as in games, puzzles and design of particular patterns. Similarities and differences between our system and other systems are almost the same, or at least very similar, so we shall discuss in more details only distinctions to Mace4.

Mace4 is developed by William McCune (1953–2011), a former professor of the University of New Mexico. According to the author, Mace4 is a program that searches for finite models of first-order theories. For a given domain size, all instances of the axioms over the domain are constructed. The result is a set of ground clauses with equality. Then, a decision procedure based on ground equational rewriting is applied. If satisfiability is detected, one or more models are printed. Mace4 is a useful complement to first-order theorem provers, with the prover searching for proofs and Mace4 looking for countermodels, and it is useful for work on finite algebras.

While Mace4 is developed for finding counterexamples (just one model), our system is designed for finding all models of a given theory. Hence, there are examples of first-order theories for which Mace4 cannot find number of all models, while our system can. Particularly, this is true for theories with axioms having several alternations of quantifiers. The reason is that Mace4 eliminates quantifiers by skolemization procedure, and thereby produces Herbrand universe. If there are several alternations, the original language is expanded and the corresponding Herbrand universe is very large. Mace4 produces models in this richer theory, i.e., original theory expanded by Skolem functions, which can be a very demanding task. Our system does not use expansions of the original theory, that is, it remains in the same language. Hence, our system directly produces models of the original theory.

The most important distinction between our system and Mace4 is the programming environment provided by the system. Mace4 is bounded to the first-order predicate logic. It means that only solutions of problems which can be stated in this logic can be evaluated. For example, it is not possible to change values of input variables. Secondly, it is also not possible to control execution of the search algorithm. This follows mostly from the fact that Mace4 is based on the backtracking algorithm. Finally, it is not easy to use it with other systems.

On the other hand, we have the full control of all mentioned aspects in our system. It is possible to change the values of variables. Actually, in our system it is possible to represent, virtually any algorithm concerning counting of finite models. We designed the system so that it is possible to use full power of programming in Python.

For example, it is possible to produce the numbers of models of the prime cardinality of any first-order theory. This is not possible in Mace4, at least not in the easy way.

The final disadvantage is that Mace4 is counting models sequentially, while our system is parallelized, which accelerates the model counting significantly. We confirmed this fact on several counting problems, as we achieved limits of some well known sequences described in the On-Line Encyclopedia of Integer Sequences [21].

Chapter 5

Applications

5.1 Lattices

In the following subsections we develop a general technique for reducing number of propositional variables by using symmetries inherent to the problem under consideration. More formally, we reduce it to the problem of determination of orbits under the action of the group of automorphisms of a finite structure on its domain. We illustrate these techniques and algorithms by writing a program in our system for a rather general problem of counting special types of partial orders and lattices.

5.1.1 Removing variables

Suppose a theory T describes a class of finite models. The set of propositional letters $\mathcal{P}$ defined by (4.3.1) and which appears in translation from T to T^* is large even for small n for domains $A = I_n$ from which $\mathcal{P}$ is generated. For example, if the language L consists of k unary relations, then $|\mathcal{P}| = kn$. If L has only one binary relation R, then $|\mathcal{P}| = n^2$. If L has only one binary operation F, then $|\mathcal{P}| = n^3$. Hence, even for small n, $\mathcal{P}$ can be enormously large. It can have hundreds, or even thousands of propositional variables. Therefore, we need a way to eliminate some propositional variables appearing in T^*. Any procedure of elimination of variables from $\mathcal{P}$ we shall call removing variables. As we have seen, the size of $\mathcal{P}$ which appears in T^* and is feasible for computing on small computers is bellow 50 and on supercomputers below 70. Let us denote by K that feasible number of variables[1]. The main goal of removing variables is to reduce T^* to a propositional theory T' having at most K variables. We note that removing variables generally results in generating an adequate set of structures, not the whole $\mathscr{L}_n$.

[1]Hence $50 \leqslant K \leqslant 70$ for today's computers

Removing variables is realized in most cases by fixing the values of certain variables. For example, if p_{ijk} represents a binary operation $i \cdot j = k, i, j, k \in A$, and if it is known that for some $a, b, c \in A$, $p_{abc} = 1$, then for all $d \in A$, $d \neq c$, we may take $p_{abd} = 0$. The next consideration explains in many cases this kind of removing variables. It is related to the definability theory and for notions and terminology we shall refer to [7].

Suppose $\mathbf{A}$ is a model of L and $X \subseteq A$. We say that X is absolutely invariant in $\mathbf{A}$ if for all $f \in \mathrm{Aut}(\mathbf{A})$, $f(X) \subseteq X$. As usual, X is definable in $\mathbf{A}$ if there is a formula $\varphi(x)$ of L so that $X = \{a \in A \colon \mathbf{A} \models \varphi[a]\}$. The proof of the next theorem is based on the Svenonius definability theorem, cf. [40], or Theorem 5.3.3 in [7].

Theorem 5.1.1. *Let* $\mathbf{A}$ *be a finite model of* L *and* $X \subseteq A$*. Then* X *is absolutely invariant in* $\mathbf{A}$ *if and only if* X *is definable in* $\mathbf{A}$*.*

Proof. Obviously, if X is definable then it is absolutely invariant. So we proceed to the proof of the other direction. In order to save on notation, we shall take $L = \{R\}$, R is a binary relation symbol. Suppose X is invariant under all automorphisms of $\mathbf{A}$. Let $\psi_1(U)$ be the following sentence of $L \cup \{U\}$, U is a new unary predicate:

$$\begin{aligned} &\forall x_1 \dots x_n \forall y_1 \dots y_n ((\bigwedge_{i<j} x_i \neq x_j \wedge \bigwedge_{i<j} y_i \neq y_j \wedge \bigwedge_{i,j} (R(x_i, x_j) \Leftrightarrow R(y_i, y_j))) \\ &\quad \Rightarrow \bigwedge_i (U(x_i) \Rightarrow U(y_i))). \end{aligned} \tag{5.1.1}$$

The sentence $\psi_1(U)$ states that U is absolutely invariant in any model $\mathbf{B}$ of L which has n elements, i.e., if $(\mathbf{B}, Y) \models \psi_1(U)$ then Y is absolutely invariant in $\mathbf{B}$.

Let ψ_2 be the following sentence of L:

$$\exists x_1 \dots x_n (\bigwedge_{i<j} x_i \neq x_j \wedge \forall x \bigvee_i x = x_i \wedge \bigwedge_{R^{\mathbf{A}}(i,j)} R(x_i, x_j) \wedge \bigwedge_{\neg R^{\mathbf{A}}(i,j)} \neg R(x_i, x_j)). \tag{5.1.2}$$

We see that the sentence ψ_2 codes the model $\mathbf{A}$, i.e., if $\mathbf{B}$ is a model of L and $\mathbf{B} \models \psi_2$ then $\mathbf{B} \cong \mathbf{A}$.

Let $\psi(U) = \psi_1(U) \wedge \psi_2$. Suppose $\mathbf{B}$ is any model of L, $(\mathbf{B}, Y)$ and $(\mathbf{B}, Y')$ are expansion of $\mathbf{B}$ to models of ψ and assume $(\mathbf{B}, Y) \cong (\mathbf{B}, Y')$. Then we see that $Y = Y'$. Therefore, by Svenonius theorem it follows that ψ defines U explicitly up to disjunction. In other words there are formulas $\varphi_1(x), \dots, \varphi_m(x)$ of L such that

$$\psi(U) \models \bigvee_i \forall x (U(x) \Leftrightarrow \varphi_i(x)) \tag{5.1.3}$$

As $(\mathbf{A}, X) \models \psi(U)$, there exists i so that $(\mathbf{A}, X) \models \forall x(U(x) \Leftrightarrow \varphi_i(x))$. Hence X is definable by $\varphi_i(x)$. □

The next corollaries follow by direct application of the last theorem to one-element absolutely invariant subsets.

Corollary 5.1.1. *Let* $\mathbf{A}$ *be a finite model of finite* L *and* $a \in A$. *If* a *is fixed by all automorphisms of* $\mathbf{A}$ *then* a *is definable in* $\mathbf{A}$ *by a formula* $\varphi(x)$ *of* L.

Corollary 5.1.2. *Let* $\mathbf{A}$ *be a finite model of finite* L. *Then* $\mathrm{Aut}(\mathbf{A}) = \{i_A\}$ *if and only if every element of* A *is definable in* $\mathbf{A}$.

Here are some examples of absolutely invariant, and hence definable subsets X in various types of finite structures $\mathbf{A}$. If $\sim$ is a relation of equivalence over A, $k \in N$, then X = "the union of all classes of equivalences of size k" is absolutely invariant. Let $\mathbf{A} = (A, \leqslant)$ be a partial order. Then the set S of all minimal elements and the set T of all maximal elements of $\mathbf{A}$ are absolutely invariant. The same holds for the set of all minimal elements of $A \backslash S$.

Our usage of definable constants for removing variables is based on the following argument. Let T be a finite theory of L and assume $\varphi_0(x), \dots, \varphi_{k-1}(x)$ are formulas of L for which T proves they are mutually disjoint, i.e., for $i \neq j$, $T \vdash \neg\exists x(\varphi_i(x) \wedge \varphi_j(x))$. Assume they define constants in T, in other words, for each i

$$T \vdash \exists x(\varphi_i(x) \wedge \forall y(\varphi_i(y) \Rightarrow \varphi_i(x))). \tag{5.1.4}$$

Let $\mathbf{B}$ be a model of T with the domain $B = I_n$. Then $\mathbf{B}$ has a unique expansion to $(\mathbf{B}, b_0, \dots, b_{k-1})$ that is a model of $T' = T \cup \{\varphi_0(c_0), \dots, \varphi_k(c_{k-1})\}$; $c_0, \dots, c_{k-1}$ are new symbols of constants to L. Since $b_i \neq b_j$ for $i \neq j$ we can define

$$f \colon I_k \to \{b_0, \dots, b_{k-1}\}, \quad f(i) = b_i, \quad i = 0, \dots, k-1. \tag{5.1.5}$$

It is easy to see that we can define labeled model $\mathbf{A}$ of L and that f extends to $h \colon (\mathbf{A}, 0, \dots, k-1) \cong (\mathbf{B}, b_0, \dots, b_{k-1})$. Hence, for an adequate set of n-models of T we can choose a set $\mathcal{K}$ of labeled models $\mathbf{A}$ of T such that $(\mathbf{A}, 0, \dots, k-1)$ is a model of T'. Therefore, models in $\mathcal{K}$ have the fixed labeling $0, \dots, k-1$ of constants definable in T.

Obviously, we can take in (5.1.5) any $S \subseteq I_n$, $|S| = k$ instead of I_k. There are $\binom{n}{k}$ such choices of S. Let s denote a permutation $s_0 \dots s_{k-1}$ of S and $\mathcal{K}_s$ the corresponding adequate set of n-models of T: for models $\mathbf{A}$ in $\mathcal{K}_s$, $(\mathbf{A}, s_0, \dots, s_{k-1})$ is a model of T'. In other words, definable elements formerly labeled by $0, \dots, k-1$ in $\mathcal{K}$ are now labeled by $s_0, \dots, s_{k-1}$ in

$\mathcal{K}_s$. Suppose S and S' are k-subsets of I_n and s, s' permutations either of S or S', $s \neq s'$. Then $\mathcal{K}_s \cap \mathcal{K}_{s'} = \emptyset$ and $|\mathcal{K}_s| = |\mathcal{K}_{s'}|$, and given that $\mathscr{L}_n = \bigcup_s \mathcal{K}_s$ we get

$$l_n = \binom{n}{k} k!|\mathcal{K}| = n(n-1)\cdots(n-k+1)|\mathcal{K}|. \tag{5.1.6}$$

In many cases theory T determines the values of atomic formulas which contains some of the definable constants. Consequently, the corresponding propositional letters from $\mathcal{P}$ have a definite value. For example, suppose R is a 2-placed relation symbol and that T proves $\forall x R(c_0, x)$. Then we can take $p_{0i} = 1$, $i = 0, \dots, n-1$. Hence, if $\mathcal{P}$ is generated over I_n, n propositional variables are killed in $\mathcal{P}$. The remaining number of variables is $n^2 - n$.

5.1.2 Definable partitions

The presented idea with definable constants can be extended to definable subsets as well. For simplicity, we shall assume that $L = \{R\}$, where R is a binary relation symbol.

A sequence $\Delta = \theta_1(x), \dots, \theta_m(x)$ of formulas of L is called a definable partition for T_n if T_n proves:

(i) $\forall x(\theta_1(x) \vee \dots \vee \theta_m(x))$.

(ii) $\neg\exists x(\theta_i(x) \wedge \theta_j(x)), \quad 1 \leqslant i \leqslant j \leqslant m$.

We shall say that Δ is a good definable partition if there are formulas $S_{ij}(x, y)$, $1 \leqslant i, j \leqslant m$, such that each $S_{ij}(x, y)$ is one of $R(x, y)$, $R(y, x)$, $\neg R(x, y)$, $\neg R(y, x)$, and T_n proves:

$$\forall xy((\theta_i(x) \wedge \theta_j(y)) \Rightarrow S_{ij}(x, y)), \quad 1 \leqslant i \leqslant j \leqslant m. \tag{5.1.7}$$

In any labeled model $\mathbf{A}$ of T_n, Δ determines a sequence $\mathcal{X}$ of definable subsets $X_1, \dots, X_m$. By a component we shall mean elements of $\mathcal{X}$. It may happen that some components are empty. The sequence of nonempty sets from $\mathcal{X} = (X_1, \dots, X_m)$ forms an ordered partition of A. A sequence $\mathcal{X}$ with this property will be called a c-partition.

Proposition 5.1.1. *Assume $|A| = n$. Then there are*

$$c_{nm} = \sum_{k=1}^{m} \binom{m}{k} \binom{n-1}{k-1} \tag{5.1.8}$$

c-partitions $\mathcal{X} = (X_1, \dots, X_m)$ of A.

Proof. Let $|X_i| = \alpha_i$. Therefore $\alpha_1, \ldots, \alpha_m$ is an integer solution of

$$n = x_1 + \ldots + x_m, \quad x_1, \ldots, x_m \geqslant 0. \tag{5.1.9}$$

Since the integer solutions of

$$n = x_1 + \ldots + x_k, \quad x_1, \ldots, x_k \geqslant 1. \tag{5.1.10}$$

are obtained from 5.1.9 by choosing k variables $x_i \neq 0$, $k \geqslant 1$, and 5.1.10 has $\binom{n-1}{k-1}$ solutions, there are $\binom{m}{k}\binom{n-1}{k-1}$ solutions of 5.1.9. Hence, there are $\binom{m}{k}\binom{n-1}{k-1}$ c-partitions $\mathcal{X}$ with exactly k nonempty sets X_i. Summing up for $k = 1, \ldots, m$, we obtain expression 5.1.8 for c_{nm}. □

Our idea for using a good definable partition Δ in generating labeled models $\mathbf{A}$ of T_n is as follows. We assume that the propositional letter p_{ij} represents $R^{\mathbf{A}}(i, j)$ as described by (4.3.6). We generate all c-partitions $\mathcal{X} = (X_1, \ldots, X_m)$ of I_n, taking that X_i corresponds to θ_i. For each $\mathcal{X}$ we assign values to particular p_{ij} in the following way. If $S(x, y)$ is $R(x, y)$ then we set $p_{ij} = 1$ for $i \in X_k$ and $j \in X_l$, $k \leqslant l$, and if $S(x, y)$ is $\neg R(x, y)$, then we set $p_{ij} = 0$. Similarly, we assign values to p_{ji} depending on whether $S(x, y)$ is $R(y, x)$ or $\neg R(y, x)$. By doing so, we obtain a propositional theory $T_{\mathcal{X}} \subseteq T_n^*$ with the reduced number of unknowns from $\mathcal{P}$.

Now, for an adequate set of models of $T_{\mathcal{X}}$ we can choose a set $\mathcal{K}_{\mathcal{X}}$ of labeled models $\mathbf{A}$ of $T_{\mathcal{X}}$ that have the following canonical labeling of components of $\mathcal{X}$

$$\begin{aligned} X_1 = \{0, 1, \ldots, \alpha_1 - 1\},\ X_2 = \{\alpha_1, \alpha_1 + 1, \ldots, \alpha_1 + \alpha_2 - 1\}, \ldots, \\ X_m = \{\sum_{i<m} \alpha_i, \sum_{i<m} \alpha_i + 1, \ldots, \sum_{i\leqslant m} \alpha_i - 1\}. \end{aligned} \tag{5.1.11}$$

Then every model $\mathbf{B} \in \mathcal{L}_{\mathcal{X}}$, where $\mathcal{L}_{\mathcal{X}}$ is the set of all labeled models of $T_{\mathcal{X}}$, is obtained from a model $\mathbf{A} \in \mathcal{K}_{\mathcal{X}}$ by choosing component X_1 from I_n, X_2 from $I_n \backslash X_1$, X_3 from $I_n \backslash \{X_1 \cup X_2\}$ and so on, until all X_i from $\mathcal{X}$ are exhausted. Therefore

$$l_{\mathcal{X}} = \binom{\beta_1}{\alpha_1} \ldots \binom{\beta_k}{\alpha_k} |\mathcal{K}_{\mathcal{X}}| \tag{5.1.12}$$

for $\mathcal{X} = (X_1, \ldots, X_k)$, $|X_i| = \alpha_i$ and

$$\beta_1 = n, \quad \beta_2 = \beta_1 - \alpha_1, \quad \ldots, \quad \beta_k = \beta_{k-1} - \alpha_{k-1}. \tag{5.1.13}$$

Let $\mathscr{P}$ be the set of all c-partitions of I_n. Note that if $\mathcal{X} \neq \mathcal{Y}$, $\mathcal{X}, \mathcal{Y} \in \mathscr{P}$, and if $\mathbf{A} \in \mathcal{K}_\mathcal{X}$ and $\mathbf{B} \in \mathcal{K}_\mathcal{Y}$, then $\mathbf{A}$ and $\mathbf{B}$ are non-isomorphic. Thus, we have

$$l_n = \sum_{\mathcal{X} \in \mathscr{P}} l_\mathcal{X}, \tag{5.1.14}$$

and

$$\kappa_n = \sum_{\mathcal{X} \in \mathscr{P}} \kappa_\mathcal{X} \tag{5.1.15}$$

where $\kappa_\mathcal{X}$ is the number of non-isomorphic models in $\mathcal{K}_\mathcal{X}$.

So, for the adequate set of models of T_n we can take the set $\mathcal{K} = \bigcup_{\mathcal{X} \in \mathscr{P}} \mathcal{K}_\mathcal{X}$ of labeled models $\mathbf{A}$ of T_n. Hence, our method for counting models of T_n is as follows. As usual, the propositional letter p_{ij} stands for $R(i, j)$.

(i) Find a good definable partition $\theta_1(x), \ldots, \theta_m(x)$ which satisfies condition (5.1.7).

(ii) Generate all c-partitions $\mathcal{X} = (X_1, \ldots, X_m)$ of I_n with canonical labeling (5.1.11).

(iii) *Removing variables*: For all $1 \leqslant k \leqslant l \leqslant m$ we fix the values of certain p_{ij} as follows. Take $p_{ij} = 1$ for $i \in X_k$ and $j \in X_l$ if $S(x, y)$ is $R(x, y)$. If $S(x, y)$ is $\neg R(x, y)$ then we take $p_{ij} = 0$. If $S(x, y)$ is $R(y, x)$, then $p_{ji} = 1$. If $S(x, y)$ is $\neg R(y, x)$, then set $p_{ji} = 0$.

(iv) Reduce T_n^* to $T_\mathcal{X}$ with the reduced number of variables using assigned values to variables p_{ij} in the previous step.

(v) Generate and count models of $\mathcal{K}_\mathcal{X}$ using $T_\mathcal{X}$ and free Boolean vectors by the procedure described in Subsection 4.3.2.

(vi) Compute $l_\mathcal{X}$ by formula (5.1.12).

(vii) Find $\kappa_\mathcal{X}$ by enumerating elements of $\mathcal{K}_\mathcal{X}$.

(viii) Repeat steps 3 through 7 until $\mathscr{P}$ is exhausted.

(ix) Compute l_n by formula (5.1.14).

(x) Compute κ_n by formula (5.1.15).

5.1.3 Irreducible components

Assume T is a finite theory. Let $\mathcal{F}$ be the set of all formulas of L having at most x as a free variable. The relation of equivalence over $\mathcal{F}$ defined by

$$\varphi \sim \psi \quad \text{iff} \quad T_n \vdash \forall x(\varphi \Leftrightarrow \psi), \quad \varphi, \psi \in \mathcal{F}, \tag{5.1.16}$$

naturally defines the Lindenbaum algebra $\mathbf{B}_{T,n}(\mathcal{F})$. Elements of $\mathbf{B}_{T,n}(\mathcal{F})$ are classes of equivalence $[\varphi]$, $\varphi \in \mathcal{F}$.

Proposition 5.1.2. $\mathbf{B}_{T,n}(\mathcal{F})$ *is finite.*

Proof. Suppose $\mathbf{B}_{T,n}(\mathcal{F})$ is infinite. Hence there is an infinite subset $S = \{\varphi_0, \varphi_1, \dots\}$ of $\mathcal{F}$ of nonequivalent formulas. Let $S^{(2)}$ be the set of all unordered pairs of different formulas from S, $\mathbf{A}_1, \dots, \mathbf{A}_m$ all unlabeled models of T_n and P the set of all nonempty subsets of $\{1, 2, \dots, m\}$. We define coloring $h\colon S^{(2)} \to P$ in the following way. Suppose $s \in S^{(2)}$, $s = \{\varphi, \psi\}$. Then

$$h(s) = \{t\colon \mathbf{A}_t \models \neg\forall x(\varphi \Leftrightarrow \psi), 1 \leqslant t \leqslant 2^m\}. \tag{5.1.17}$$

We see that $h(s) \neq \emptyset$ since $\neg\forall x(\varphi \Leftrightarrow \psi)$ is consistent with T_n, hence there is $\mathbf{A}_i$ which satisfies this sentence. By Ramsey's theorem there is an infinite $Y = \{\psi_0, \psi_1, \dots\}$, subset of S, which is homogeneous for coloring h. Let $t \in h(\{\psi_0, \psi_1\})$ and $X_i \subseteq A_t$, $i = 0, 1, \dots$, definable in $\mathbf{A}_t$ by ψ_i. Then $X_i \neq X_j$ if $i \neq j$, so A_t would be a finite set with an infinite number of different subsets, a contradiction. □

Since $\mathbf{B}_{T,n}(\mathcal{F})$ is finite, it has the finite number of atoms, say $[\theta_1], \dots, [\theta_m]$. We see that θ_i cannot be split into two disjoint consistent formulas of $\mathcal{F}$. In other words, there are no $\psi \in \mathcal{F}$ such that $T_n \vdash \forall x(\psi \Rightarrow \theta_i)$ and each of $\exists x\psi$ and $\exists x(\theta_i \wedge \neg\psi)$ is consistent with T_n. Such θ_i will be called an irreducible formula of T_n. Suppose $\mathbf{A}$ is a model of T_n. Then every irreducible formula θ defines a subset X_θ in $\mathbf{A}$. If X_θ is nonempty, then it cannot be split into two disjoint nonempty definable subsets, i.e., X_θ does not contain a proper nonempty definable subset. A nonempty subset of $\mathbf{A}$ definable by an irreducible formula is called an irreducible component of $\mathbf{A}$.

Proposition 5.1.3. *Suppose* $\mathbf{A}$ *is a model of* T_n *and* $\mathcal{X}$ *is the set of all irreducible components of* $\mathbf{A}$. *Then* $\mathcal{X}$ *is a partition of* A.

Proof. Atoms of $\mathbf{B}_{T,n}(\mathcal{F})$ make a partition of $\mathbf{1}$, so the statement follows. □

We can estimate the number of such partitions by (5.1.8), given that $\mathcal{X}$ is also a c-partition, albeit an irreducible one.

Furthermore, ultrafilters of $\mathbf{B}_{T,n}(\mathcal{F})$ are determined by types of T_n. Hence, an atom of $\mathbf{B}_{T,n}(\mathcal{F})$ is $[\theta]$ where θ is the conjunction of formulas from a type $t(x)$ of T_n. In this way one can obtain all irreducible formulas of T_n. However, there is a more useful way of describing irreducible formulas and components as we will show in the following.

Let $\mathbf{A}$ be a model of T_n with $A = I_n$ and $G = \mathrm{Aut}(\mathbf{A})$ and $H\colon G \to \mathrm{Sym}(A)$ defined by $H(g)(x) = g(x)$, $g \in G$, $x \in A$. Then H is a group action of G on A and orbits of this action are $x^G = \{g(x)\colon g \in G\}$, $x \in A$.

Proposition 5.1.4. *Let T be a finite theory of L. Then the irreducible components in a model $\mathbf{A}$ of T_n are exactly the orbits under the action H.*

Proof. Let $\theta(x, y)$ denote the following formula:

$$\exists x_1 \ldots x_n \exists y_1 \ldots y_n (\bigwedge_{i<j} x_i \neq x_j \wedge \bigwedge_{i<j} y_i \neq y_j \wedge \\ \bigwedge_{i,j} (R(x_i, x_j) \Leftrightarrow R(y_i, y_j)) \wedge \bigvee_i (x = x_i \wedge y = x_i)). \tag{5.1.18}$$

Then $\theta(x, y)$ expresses that x and y have the same orbits under action H, while $U(x)$ defined by $\forall y(U(y) \Leftrightarrow \theta(x, y))$ says that x belongs to an orbit. Hence, the sentence $\sigma(U)$ of $L \cup \{U\}$

$$\forall x \forall y(U(y) \Leftrightarrow \theta(x, y)) \tag{5.1.19}$$

implicitly defines the notion of orbit. Suppose $(\mathbf{A}, X)$ and $(\mathbf{A}, Y)$ are models of $T_n + \sigma(U)$ and $f\colon (\mathbf{A}, X) \to (\mathbf{A}, Y)$. Hence $f \in \mathrm{Aut}(\mathbf{A})$ and for some $a \in A$, $X = \{g(a)\colon g \in G\}$ and so $Y = \{f(g(a))\colon g \in G\} = \{g(a)\colon g \in G\} = X$. Therefore, conditions of Svenonius theorem are fulfilled, so $T_n + \sigma(U)$ defines U explicitly up to the disjunctions. Hence, the orbit of each element of $\mathbf{A}$ is definable in $\mathbf{A}$.

Let $X \subseteq A$ be the orbit of an element of $\mathbf{A}$ an suppose that X contains a proper nonempty definable subset Y. Then there are $a \in Y$ and $b \in X$. Since a and b belong to the same orbit, there is $g \in \mathrm{Aut}(\mathbf{A})$ such that $g(a) = b$. On the other hand $g(Y) \subseteq Y$ since Y is definable, a contradiction. Therefore, X is an irreducible component. □

Note 5.1.4.1 When we started working on the graph isomorphism problem, we were not fully aware of this group theoretical nature of our approach. It was only after we researched some of the state-of-the-art algorithms [28] that we realized the existence of this correspondence between types and orbits.

Our general idea for generating labeled models of a finite theory of a language with one binary relation symbol is to use the following proposition for removing variables.

Proposition 5.1.5. *Let T be a finite theory of $L = \{R\}$, R is a binary relation symbol, and let $S(x, y)$ be one of $R(x, y)$, $R(y, x)$, $\neg R(x, y)$, $\neg R(y, x)$. Assume that $\theta(x)$ is an irreducible formula of T_n and $\varphi(x)$ is a formula of L. Suppose*

$$T_n \vdash \exists x(\theta(x) \wedge \forall y(\varphi(y) \Rightarrow S(x, y))). \tag{5.1.20}$$

Let $\mathbf{A}$ be a model of T_n, X_θ the component of $\mathbf{A}$ associated to θ and X_φ the subset definable in $\mathbf{A}$ by φ. Then $\mathbf{A} \models S[a, b]$ for every $a \in X_\theta$ and every $b \in X_\varphi$.

Proof. Let $\psi(x)$ denote the formula $\theta(x) \wedge \forall y(\varphi(y) \Rightarrow S(x, y))$. By condition 5.1.20, in $\mathbf{B}_{T_n}(\mathcal{F})$ we have $[\psi] \leqslant [\theta]$ and $[\psi] \neq \mathbf{0}$. But $[\theta]$ is atom in $\mathbf{B}_{T_n}(\mathcal{F})$, hence $[\theta] = [\psi]$. Therefore, $T_n \vdash \theta(x) \Rightarrow \forall y(\varphi(y) \Rightarrow S(x, y))$, hence $a \in X_\theta$ implies $S^{\mathbf{A}}(a, b)$ for all $b \in X_\varphi$. □

We note that there are several variations on how this can be done, but the main ones are as follows. We assume that the propositional letter p_{ij} represents $R^{\mathbf{A}}(i, j)$ as described by (4.3.6). Suppose $\theta_1(x), \dots, \theta_m(x)$ is the sequence of all irreducible formulas. We generate all partitions $\mathcal{X} = (X_1, \dots, X_m)$ of I_n, which are potentially orbits of $\mathbf{A}$ that we want to generate, taking that X_i corresponds to θ_i. This assumption is consistent with Proposition 5.1.4. Now suppose we found θ_l and $\varphi(x)$ for which condition (5.1.20) holds. Since $[\theta_i]$ are atoms, φ is a disjunction of some irreducible formulas, thus $X_\varphi = X_{i_1} \cup \ldots \cup X_{i_k}$ for some choice of elements of the partition $\mathcal{X}$. If $S(x, y)$ is $R(x, y)$ then we set $p_{ij} = 1$ for $i \in X_l$ and $j \in X_\varphi$ and if $S(x, y)$ is $\neg R(x, y)$, then we set $p_{ij} = 0$. Similarly, we assign values to p_{ji} depending on whether $S(x, y)$ is $R(y, x)$ or $\neg R(y, x)$. Therefore, we reduced the number of unknowns from $\mathcal{P}$, and now we can generate the adequate set of labeled models in a way similar as before.

On the other hand, if the condition (5.1.20) is not fulfilled, we can refine the procedure in the following way. Let us call a pair (i, j) an arrow if $\mathbf{A} \models S[i, j]$, $i, j \in I_n$. For an arrow (i, j) we say it is an arrow from θ_l to φ if $i \in X_l$ and $j \in X_\varphi$. Similarly to the proof of Proposition 5.1.5, one can prove the following: if $\mathbf{A}$ is a model of T_n, and if for some $i \in X_l$ there are exactly s arrows from θ_l to φ, then for every $i \in X_l$ there are exactly s arrows from θ_l to φ. This is in fact obvious since the elements of an orbit are indiscernible. Therefore, for every model $\mathbf{A}$ of T_n there is $s \in I_n$ such that for every $i \in X_l$ there are exactly s arrows (i, j) from θ_l to φ. Hence, we can set the appropriate values to p_{ij}. For instance, if $S(x, y)$ is $R(x, y)$, then we take $p_{ij_1}, \dots, p_{ij_s} = 1$ where $i \in X_l$ and $j_1, \dots, j_s$ are different elements from X_φ. For the rest of $j \in X_\varphi$ we take $p_{ij} = 0$. And again, we can proceed in a similar manner as we did earlier.

Example 5.1.1. Let T be the theory of partial orders of $L = \{\leqslant\}$ having at least 2 elements with extra axioms that state: there are the least and the greatest element. Instead of T we can take the theory T' of partially ordered sets which are upward and downward directed. Theories T

and T' are not equivalent, as T' has an infinite model which is not a model of T. But T and T' have the same finite models.

We see that $l_{T,n} = n(n-1)|\mathcal{K}|$, $n \geqslant 2$, where $\mathcal{K}$ is the set of all partial orders $\mathbf{A} = (A, \leqslant, 0, 1)$, $A = I_n$; 0 is the least and 1 is the greatest element in $\mathbf{A}$. Since p_{ij} states $i \leqslant j$ and $\leqslant$ is reflexive, we can also take

$$\begin{aligned} &p_{0i} = 1,\ p_{j0} = 0,\ p_{i1} = 1,\ p_{1k} = 0,\ p_{ii} = 1, \\ &i = 0, \ldots, n-1,\ j = 1, \ldots, n-1,\ k = 0, \ldots, n-2. \end{aligned} \tag{5.1.21}$$

Hence, $5n - 6$ variables are killed and T^* is reduced to T' which has $v = n^2 - 5n + 6$ variables. If $n = 8$ then $v = 30$ and all partial orders having 8 elements are generated at one go on our hardware configuration. By adding to T some new axioms, we can generate models of a new theory in the same way and for the same time. For example, we can compute all lattices of order 8 by adding just one axiom to T.

With small adjustments, this algorithm works in real time for $n \leqslant 10$. Namely, for larger n, the feasibility constant K, see the footnote 1 (page 73), is exceeded. So we have to use the previously described procedure based on components. Therefore, let us define recursively the following sequence of length n of the following formulas

$$\theta_0(x) \equiv \forall y(x \leqslant y), \quad \theta_{k+1}(x) \equiv \forall y(\bigvee_{i \leqslant k} \theta_i(y) \vee \neg\, y < x) \wedge \bigwedge_{i \leqslant k} \neg\, \theta_i(x). \tag{5.1.22}$$

Theorem 5.1.2. *Formulas $\theta_k(x)$ make a good definable partition in T_n.*

Proof. If $\mathbf{A} = (A, \leqslant)$ is a partial order, we see that the associated components are: $X_0 = \{0\}$, 0 is the least element of $\mathbf{A}$, X_1 is the set of minimal elements of $A \backslash \{0\}$, X_2 is the set of minimal elements of $A \backslash (X_0 \cup X_1)$, and so on. Let us call an element of X_k, a k-minimal element. Since $X_{i+1} \neq \emptyset$ implies $X_i \neq \emptyset$, we see that $X_k = \emptyset$ for $k > n$.

Since for every $k \leqslant n-2$ there is a model of T which has $X_k \neq \emptyset$, we see that $[\theta_k] \neq \mathbf{0}$ in the Lindenbaum algebra $\mathbf{B}_{T_n}(\mathcal{F})$. Note that the formulas θ_k make partition of $\mathbf{1}$ in $\mathbf{B}_{T_n}(\mathcal{F})$. Let $[\varphi]$ be an atom of $\mathbf{B}_{T_n}(\mathcal{F})$. □

5.1.4 Lattice counting: program implementation

The problem of exact counting of special types of lattices took attention of many authors. For example, at OEIS there are several hundreds articles related to various counting problems on lattices. Here we describe a general structure of the program for counting lattices and other types of partial orderings based on the theory developed in the previous subsections:

Preamble Contains the required imports, which can be either the usual Python imports, or the ones related to our system (e.g., `GPURunner`).

Domains Definition of domains. In order to reduce the size of formulas and the number of computational steps we introduce special domains. For example, instead of using the set of all functions $f: n \to n$ we use the set of permutations S_n.

Axioms In this part we write the axioms in first-order logic for the class of structures that we are interested in. Depending on the axiom, we can write it using our DSL, construct it in an imperative manner, or simply make the equivalent assumptions as we did for reflexivity.

Main In this part we implement the counting of the structures. Therefore, the crucial role in this part is played by hardware accelerated primitives from our system. For example, we used the `count` primitive to accelerate counting using a GPU.

The full listing of the program with the explanations is included in §A.2.

5.2 Tournament graphs

Theory of random graphs lies on the border of combinatorics, probability theory and mathematical logic. This theory was created in the sixties of the last century in the works of Erdös – Rényi. For the subsequent development of this theory are also credited Béla Bolobás, Noga Alon and Joel Spencer.

The goal of this section is to introduce the main ideas of this theory, to give some constructions and to present some applications of our system on these graphs. We shall also give applications of the famous 0–1 law for finite structures (Glebski, Kogan, Liagonki i Talanov [1969], Fagin [1976]) in this area.

Theorem 5.2.1 (0–1 Law)**.** *If φ is a first order property in the language of graphs, then one of the statements φ, $\neg\varphi$ is true on almost all finite graphs.*

5.2.1 Graphs and tournament

Simple oriented graph is a structure $\mathbf{G} = (G, R)$, where R is a binary relation which satisfies

$$R(x, y) \Rightarrow \neg R(y, x), \quad x, y \in G$$

Simple undirected graph is a graph $\mathbf{G} = (G, R)$ in which the binary relation R is symmetric. In graph theory, by graphs usually are assumed undirected graphs.

An element of the domain G is called vertex or node. An edge is associated with two vertices, and the association takes the form of the ordered pair of the vertices. Hence, an edge is a pair $(x, y) \in R$. In undirected graphs, instead of the binary relation R the set of two-element subsets of G is taken. In this case the relation R is called adjacency relation.

In a complete graph for every different $x, y \in G$, x and y are adjacent. We see that in an n-element G there are $n(n-1)/2$ edges. Also every two n-elements complete graphs are isomorphic.

A tournament is a complete oriented graph. On a set S with n elements there are $2^{\frac{n(n-1)}{2}}$ different tournaments.

Hence, tournaments are described by the following axioms:

(i) $x < y \vee y < x$, (completeness).

(ii) $x < y \Rightarrow \neg(y < x)$, (directedness).

5.2.2 Kauer's graph

A binary relation $<$ on a finite domain A is a Kauer's graph if it is asymmetric, dense and without upper bound. Axioms for these graphs are:

(i) $x < y \Rightarrow \neg(y < x)$, asymmetric.

(ii) $\forall xy \exists z(x < y \Rightarrow x < z < y)$, dense.

(iii) $\forall x \exists y(x < y)$, without upper bound.

Hence, Kauer's relation is anti-reflexive, $\forall x \neg(x < x)$. Kauer's tournament is a finite Kauer's graph which is also a tournament. Unbounded Kauer's tournament is a Kauers's tournament without a minimal element.

We encountered this type of graphs while studying a software for finding finite models Mace4 [27]. There was asked a following question:

(i) Is there a finite Kauer's graph? If there is one, what is the the least possible order (the number of vertices) of an Kauer's graph? (n=7)

This question is very easily answered by both Mace4 and our system, even though it is not as easy for human to do so. We also posed the following questions:

(ii) For which positive integers n there is a Kauer's tournaments of order n. (All $n \geqslant 7$).

(iii) What is the number k_n of nonisomorphic unbounded tournaments of order n. ($\lim_{n\to\infty} k_n = \infty$)

In answering these questions we needed a notion of a random graph.

5.2.3 Random graphs with n vertices

Random graphs with vertices $1, 2, \ldots, n$ are generated in the following way:

> For every pair of elements i, j, $i < j$, a coin is tossed. If the head turned up, then we take $i \sim j$. Otherwise, i.e., the tail was up, the vertices i and j are not adjacent.

In this way a discrete probability space is determined with the binomial distribution having the parameters n and $p = 1/2$. In general, for arbitrary but fixed $0 < p < 1$ we have a discrete probability space with the binomial distribution having the parameters n and p. For creation of this probability space it suffices to arrange the tossing of the coin so that it falls with the probability p. In this way a random graph $G(n, p)$ is obtained. We see that:

$$\text{The probability of } i \sim j \text{ is } \Pr[i \sim j] = p. \tag{5.2.1}$$

Using the above formula, we see that if H is a randomly generated graph with k vertices, then

$$\Pr[\{H\}] = p^k (1 - p)^{\binom{n}{2} - k}. \tag{5.2.2}$$

Therefore, the elementary event for an n-element graph H is:

"The graph H has exactly k vertices".

5.2.4 Random tournaments

Random tournaments are generated in a similar way as random graphs: a coin is tossed, then if the head turned up we take $i \prec j$, otherwise we take $i \succ j$. The probability space has the binomial distribution with parameters n, $1/2$. Since we assumed the binomial distribution, the elementary events $i \sim j$, respectively $i \prec j$ are independent.

More general case for random graphs $G(n, p)$, when p is not constant, is also considered. For example, it is taken $p = p(n)$ where n is the order of the graph. It is also assumed that $\lim_{n \to \infty} p(n) = 0$.

5.2.5 Example 1: Erdös, [1963]

For historical reasons, as a first example, we give one of the first problems on random graphs which was solved by P. Erdös. This example also illustrates techniques used for solving problems on randomly generated graphs.

Let $\mathbf{G}$ be a tournament with n players (vertices). We say that $\mathbf{G}$ has the property $\mathbb{S}_k$ iff for any choice of k players there is a player on the tournament who won all the players from this

choice. In other words, for any $a_1, a_2, \dots, a_k \in G$ there is a $b \in G$ such that:

$$a_1 \prec b,\ a_2 \prec b,\ \dots,\ a_k \prec b.$$

For example:

$\mathbb{S}_1,\ (n = 3):\quad a \prec b \prec c \prec a.$

$\mathbb{S}_2,\ (n = 7):\quad$ Kauer's graph with 7 vertices.

Problem (Schütte)**.** Is it true that for every k there is a tournament with the property $\mathbb{S}_k$.

Theorem 5.2.2 (Erdös)**.** *If*

$$\binom{n}{k}\left(1 - \frac{1}{2^k}\right)^{n-k} < 1, \tag{5.2.3}$$

then there is a tournament with n players and the property $\mathbb{S}_k$.

Proof. Suppose (5.2.3) and let $\mathbf{G} = G(n, \frac{1}{2})$ be a random tournament. Further, let $K \subseteq G$ be with k elements and $v \in G \setminus K$. Then

$$\begin{aligned}
&A_K = \text{“There is no player in } G \setminus K \text{ who won all the players from } K\text{”},\\
&\Pr[\text{“}v \text{ won all players from } K\text{”}] = 1/2^k,\\
&\Pr[\text{“}v \text{ did not win all players from } K\text{”}] = 1 - 1/2^k,\\
&\Pr[\text{“There is no player in } G \setminus K \text{ who won all the players from } K\text{”}]\\
&\qquad = \Pr[A_K] = (1 - 2^{-k})^{n-k},\\
&\Pr[\text{“There is } K \text{ such that } A_K\text{”}] = \Pr\left[\textstyle\bigvee_{K \subseteq G} A_K\right] = \Pr\left[\textstyle\bigcup_{K \subseteq G} A_K\right]\\
&\qquad \leqslant \sum_{K \subseteq G} \Pr[A_K] \leqslant \binom{n}{k}\left(1 - \frac{1}{2^k}\right)^{n-k} < 1,\\
&\Pr\left[\textstyle\bigwedge_{K \subseteq G} \neg A_K\right] > 0.
\end{aligned}$$

That is, there is a tournament with n vertices and the property $\mathbb{S}_k$. □

5.2.6 Example 2: Kauer's tournament

Theorem 5.2.3 (Mijajlović–Pejović)**.** *For all $n \geqslant 7$ there is an unbounded Kauer's tournament.*

Proof. Let $n \geqslant 20$ and $G(n, \frac{1}{2})$ be a random tournament. Let for a two-element subset $K = \{i, j\} \subseteq G$ the event

$$A_K = \text{“There is no } k \in G \setminus K,\ i < k < j\text{”}.$$

Similarly as in the previous example we find

$$\Pr[A_k] = \frac{1}{2}\left(1 - \frac{1}{2^2}\right)^{n-2}, \quad \Pr[\bigvee_{K \subseteq G} A_K] \leqslant \frac{1}{2}\binom{n}{2}\left(1 - \frac{1}{2^2}\right)^{n-2}.$$

Also, $\Pr[\neg\forall i \exists j \ \ i < j] = n/2^{n-1}$.

For $n = 7, 8, \dots, 19$ Kauer's graphs are directly generated by use of a computer. Whereas, for $n \geqslant 20$ it is true that

$$\frac{1}{2}\binom{n}{2}\left(1 - \frac{1}{2^2}\right)^{n-2} + \frac{n}{2^{n-2}} < 1,$$

so as in the previous example we find an unbounded Kauer's graph with n vertices. □

5.2.7 Supplement A: 0–1 Law

We used essentially 0–1 law in proving some properties of Kauer's graphs and answering questions (ii) and (iii). So we decided to include a subsection on 0–1 law, probably the most important theorem in finite model theory, for the completeness reasons, and for the comfortness of a reader of this thesis. Presented proofs are often only outlined.

Rado graph

For a sentence θ of graph theory let $\Pr_n(\theta)$ denote the probability of the event:

> "θ is true in a randomly generated graph of order n".

Let k and l be natural numbers and consider the following sentence on graphs:

> ψ_{kl} = "For every choice of different edges $x_1, \dots, x_k$ and $y_1, \dots, y_l$ there is a different vertex z which is connected with all vertices x_i and it is not connected with any of the vertices y_j".

Then for $m = k + l$ the following holds

$$c_n = \Pr_n[\neg\psi_{kl}] \leqslant \binom{n}{k}\binom{n-k}{l}\left(1 - \frac{1}{2^m}\right)^{n-m}.$$

For fixed k and l, the term $\binom{n}{k}\binom{n-k}{l}$ is a polynomial in the indeterminate n of the degree $m = k + l$. On the other hand the term $(1 - \frac{1}{2^m})^{n-m}$ is an exponential function in the variable n having the base $0 < 1 - \frac{1}{2^m} < 1$. Hence, $\lim_{n\to\infty} c_n = 0$ and therefore every finite subset of the theory

$$T = \{\psi_{kl} \colon k, l \in N\}$$

is consistent and all models of T are infinite.

Definition 5.2.1. Rado graph is every countable model of the theory T.

Example 5.2.1 (Ackermann, Rado)**.**

(i) $(V_\omega, \sim)$, $x \sim y$ iff $x \epsilon y$ or $y \epsilon x$. Here, $V_\omega = \cup_n V_n$, where

$$V_0 = \emptyset, \quad V_{n+1} = V_n \cup P(V_n),\ n = 0, 1, \ldots .$$

If $x_1, x_2, \ldots, x_k, y_1, y_2, \ldots, y_l$ are different elements of V_ω then $z = \{x_1, x_2, \ldots, x_k, \{y_1, y_2, \ldots, y_l\}\}$ is adjacent to all x_i but not to any y_i.

(ii) Let the domain of the graph be the set of natural numbers N. The graph relation $\sim$ for $i, j \in N$ is defined by:

$$i \sim j \text{ iff } i \text{ occurs in the binary expansion of } j \text{ or vice versa.}$$

Properties of Rado graph

Theorem 5.2.4. *The theory T is ω-categorical, i.e., every two Rado graphs are isomorphic.*

Proof. Back–and–forth (zig–zag) argument. □

An immediate consequence of the preceding theorem is the following corollary.

Corollary 5.2.1. *The theory T is complete.*

One more corollary is:

Corollary 5.2.2. *Rado graph is a saturated model.*

Theorem 5.2.5. *Rado graph is universal, i.e., every at most countable graph can be embedded in a Rado graph.*

Proof. This property follows from the saturation of the Rado graph, or it can be proved directly by one half of the zig–zag argument. □

0–1 Law

Theorem 5.2.6 (Fagin – GKLT)**.** *Let* **R** *be a Rado graph and φ a sentence of graph theory. Then*:

$$\mathbf{R} \models \varphi \text{ iff } \varphi \text{ is true in almost all finite graphs.}$$

Proof.

($\Rightarrow$) $\mathrm{Th}(\mathbf{R}) = T$ since T is a complete theory. Therefore, if $\mathbf{R} \models \varphi$ then $T \models \varphi$, i.e.,

$$T \vdash \varphi, \text{ so for some } \theta_1, \ldots, \theta_m \in T,$$
$$\vdash \bigwedge_{i \leqslant m} \theta_i \Rightarrow \varphi, \text{ that is } \vdash \neg\varphi \Rightarrow \bigvee_{i \leqslant m} \neg\theta_i .$$

Since for all indices i it holds $\lim_{n\to\infty} \mathrm{Pr}_n[\neg\theta_i] = 0$, it follows

$$\mathrm{Pr}_n[\neg\varphi] \to 0, \quad \text{as} \quad n \to 0.$$

($\Leftarrow$) This direction follows from Compactness theorem and the categoricity of T. □

Corollary 5.2.3. 0–1 *Law.*

5.2.8 Supplement B: Further problems

There are also problems widely considered in the literature on finite graphs similar to the problems (i), (ii), and (iii) stated for Kauer's graphs. Therefore, we are also interested in studying problems from the following list, some of which we have already considered and partially solved:

Problem (Existence of Graph Isomorphism) Given two graphs G and G' with n vertices each, decide whether they are isomorphic.

Problem (Existence of Isomorphism of Labeled Graphs) Given two labeled graphs $(G, \mathbf{a})$ and $(G', \mathbf{a}')$, decide whether they are isomorphic.

Problem (Graph Isomorphism) Given two graphs G and G', decide whether they are isomorphic, and if so, construct an isomorphism from G to G'.

Problem (Graph Automorphism) Given a graph G, determine a generating set for $\mathrm{Aut}(G)$.

Problem (Order of the Automorphism Group) Given a graph G, determine the order of $\mathrm{Aut}(G)$.

Problem (Number of Isomorphisms) Given two graphs G and G' with n vertices each, determine the number of isomorphisms from G to G'.

Chapter 6

Conclusion

The most results of the dissertation have been stated immediately after their presentation in the relevant section, so here we give only a brief overview of all the results obtained. Finally, we briefly discuss possible extensions of the developed software from both a theoretical and a practical standpoint.

6.1 Summary of results

The main and original results of this doctoral dissertation are presented in Chapters 3, 4 and 5.

In Chapter 3, an original application of Hopf algebras in the derivation of Gould inversion formulas [16] [17] is presented. We point out an interesting case for $n = 2$, where we used a nice symmetry property of Tchebychev polynomials. In addition, an original generalisation of the mentioned inversion formulas is given. These results are published in [31].

In Chapter 4, a new proof of Birkhoff HSP theorem from the field of universal algebras is presented. This proof is based on the newly introduced concept of interpretation of variables. Namely, the concept of a variable assumes a domain D in which this variable takes values. These assignments enable us to compute the values of formulas in a given operational-relational structure having D as a domain. Our idea was to interpret a variable v as a map $\hat{v}$ having valuations μ over D as arguments, while the values are the elements of D computed by applying the evaluation operator, $\hat{v}(\mu) = \mu(v)$. In this way, we can naturally define a notion of free generators of algebras, what enabled us to find an elegant proof of the HSP theorem.

In the continuation of this chapter, following the ideas of Mijajlović et al. [30] [33], we developed a method for computing the number of finite models of first-order theories using finite free Boolean algebras. The method itself consists of several precisely defined procedures that were later fully implemented. The first step of the method is the translation of formulas of the first-order predicate calculus into the propositional formulas. As the method refers to finite

structures, a set of axioms for a model is translated into a finite set of propositional formulas. The procedure itself consists of indexing propositional variables with domain elements, which corresponds to naming, i.e., denotation of domain elements by symbols of constants. The next procedure that is elaborated in the thesis is to compute the values of logical formulas based on the valuations of propositional variables. It turns out that the valuations of propositional variables are cross-sections of free vectors. This naturally led to the next step, the generation of free vectors of finite Boolean algebras. This step, as well as computing the logical values of propositional formulas, were implemented by parallel programming in OpenCL and Python. With this procedure, we can determine all the valuations that make a given set of propositional formulas true, that is, all models. Each propositional model corresponds to exactly one model of the initial first-order theory whose translation we made in the first step.

On these bases, the main result of the thesis was developed, which is a parallel software system for generating and counting models (operational-relational structures) of first-order theories. The system was implemented in Python programming language using hardware accelerated primitives implemented in OpenCL. This enables the system to use all available hardware resources of both the GPU and the CPU when computing the logical values of propositional formulas. The system itself consists of several separate but related features. First of all, there are four hardware accelerated primitives:

(i) `eval(expression)`
Computes all models of the given `expression` using parallel evaluation.

(ii) `count(expression)`
Computes the number of models of the given `expression` using parallel counting.

(iii) `findmodel(expression)`
Computes a model of the given `expression` using parallel search.

(iv) `exist(expression)`
Returns `True` if the given `expression` has a model, `False` otherwise. In other words, it answers whether the given theory is consistent.

These primitives are integrated into the Python environment and can be used to examine theories from a model-theoretical point of view. In addition to hardware accelerated primitives, which are the main feature of the system, we also developed features such as:

- A domain-specific language (DSL) for formulating axioms of first-order theories; which facilitates the input of theories into the system,

- Their translation into propositional calculus; which is the implementation of the first step of our method for computing the number of finite models of first-order theories, and
- Partial evaluation of propositional formulas; which is necessary in the process of elimination of propositional variables.

It should be pointed out that all the described functionalities are integrated into the Python environment in a similar way to the functionalities of other Python libraries, which enables very convenient use of the system.

In Chapter 5, a procedure is developed for the computer counting of finite partially ordered sets, especially lattices, of given cardinality. It is also proved that for every $n \geqslant 7$ there exists a Kauer's graph with n vertices. This proof relies on the theory of random graphs and the 0–1 law for finite models, while the generation of several computationally difficult cases necessary for the proof was done using our system. Both examples have common features such as the elimination of variables, the computation of definable partitions, and the computation of orbits under group action which in some way is a measure of the symmetry of a structure.

6.2 Related work

There are several software systems that are used for model generating and counting. They range from general finite model finders like Mace4 [27] and Paradox [8], to more specialized ones like PSATO [42] which was developed for computing specific quasigroups. Our system belongs to the group of general systems.

It is shown in the thesis how our software can be used to solve elementary problems in combinatorics. On the other hand, using the system to solve more difficult problems, as is the case with the examples in Chapter 5, involves a solid knowledge of the theoretical background of the problem. The main theoretical aspect relates primarily to the representation of a computational problem in the first-order predicate calculus. Another important aspect is the reduction of the number of propositional variables. This problem can be solved, for example, using definability theory and parts of classical model theory.

A similar requirement, of knowing the theoretical background of the problem, is also encountered in the other mentioned systems. What sets our system apart from others is the way it is implemented, as well as the principles on which that implementation is based. Our system is based on a method developed using finite free Boolean algebras, whereas, for instance, Mace4 is based on skolemization of first-order theories. Therefore, Mace4 generates models with redundant objects (Skolem functions), which interferes with and complicates the counting of models of a theory. In addition, most of these systems are standalone programs, so it is not

straightforward to use them from a programming environment. On the other hand, our system was designed from the ground up to be used as a Python library.

There is a rich literature on applications of these systems and algorithms in solving problems in finite combinatorics, finite models, as well as in games, puzzles and design of particular patterns. Many of these problems can also be solved in our system. For example, we first became acquainted in Mace4 with the problem of the existence of Kauer's graphs for the case $n = 7$. We solved this problem in the thesis for an arbitrary natural number n.

6.3 Future work

There are several directions for further development of our system. One of the most interesting options is the introduction of arithmetic operations on indices of propositional variables. For example,

$$p_i \wedge p_j \Rightarrow p_{i+j}.$$

This new expressiveness would allow some combinatorial problems to be expressed and solved more naturally.

A similar innovation would be the introduction of quantifier restrictions using binary relations on their domains such as, say, some partial ordering $\leqslant$:

$$\bigwedge_{i \leqslant j} p_{ij}$$

It would also be interesting to add full support for the use of branching or Henkin quantifiers, which are currently only partially supported.

It is expected that further performance improvement of the system is possible. This is primarily related to the acceleration of parallel computation of propositional formulas. Better integration of the system into the Python environment is also expected, as well as the ability to use multiple GPUs.

References

[1] M. Abramowitz and I. A. Stegun. *Handbook of mathematical functions*. U.S. Dept. of Standards, National Bureau of Standards, 10th printing edition, 1972.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2nd edition, 2007.

[3] R. J. Bayardo Jr. and J. D. Pehoushek. Counting models using connected components. In *Proceedings of AAAI-00: 17th National Conference on Artificial Intelligence*, pages 157–162, 2000.

[4] A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability*. IOS Press, 2009.

[5] J. Blasiak. Nonstandard braid relations and chebyshev polynomials. arXiv:1010.0421v1 [math.RT], 2010.

[6] S. Burris and H. P. Sankappanavar. *A course in Universal algebra*. Springer, 2012 update edition, 1981.

[7] C. C. Chang and J. H. Keisler. *Model theory*. North Holland, 1990.

[8] K. Claessen and N. Sörensson. Paradox. URL http://www.cs.chalmers.se/~koen/paradox/.

[9] F. A. Costabile and E. Longo. An algebraic exposition of umbral calculus with application to general linear interpolation problem – a survey. *Publ. Inst. Math. (Belgrade)*, 96(110): 67–83, 2014.

[10] A. Darwiche. New advances in compiling cnf into decomposable negation normal form. In *Proceedings of ECAI-04: 16th European Conference on Artificial Intelligence*, pages 328–332, 2004.

[11] S. Dascalescu, C. Nastasescu, and S. Raianu. *Hopf Algebra: An Introduction*. CRC Press, 2000.

[12] R. J. Duffin, E. L. Peterson, and C. Zener. *Geometric Programming*. J. Wiley and Sons, 1967.

[13] M. Filaseta, F. Luca, P. Stãnicã, and R. G. Underwood. Galois groups of polynomials arising from circulant matrices. *Jour. Number Theory*, 128(1):59–70, 2008.

[14] M. Fowler. *Domain-Specific Languages*. Addison-Wesley, 2010.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[16] H. W. Gould. A new convolution formula and some new orthogonal relations for inversion of series. *Duke Mathematical Journal*, 29(3):393–404, 1962.

[17] H. W. Gould. New inverse series relations for finite and infinite series with applications. *Jour. Math. Research and Exposition*, 4(2):119–130, 1984.

[18] H. W. Gould. The girard-waring power sum formulas for symmetric functions and fibonacci sequences. *Fibonacci Quart.*, 37(2):135–140, 1999.

[19] H. W. Gould. The inverse of a finite series and a third-order recurrent sequence. *Fibonacci Quart.*, 44(4):302–315, 2006.

[20] Y. Hamadi and C. M. Wintersteiger. Seven challenges in parallel sat solving. In *Proceedings of AAAI-12: 26th AAAI Conference on Artificial Intelligence*, pages 2120–2125, 2012.

[21] OEIS Foundation Inc. The on-line encyclopedia of integer sequences. URL http://oeis.org/.

[22] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[23] C. Jordan. *Calculus of finite difference*. Chelsea Publ. Comp., N.Y., 1950.

[24] G. Katsirelos, A. Sabharwal, H. Samulowitz, and L. Simon. Resolution and parallelizability: barriers to the efficient parallelization of sat solvers. In *Proceedings of AAAI-13: 27th AAAI Conference on Artificial Intelligence*, pages 481–488, 2013.

[25] W. Lang. On sums of powers of zeros of polynomials. *Jour. of Comput. and Appl. Math.*, 89(2):237–256, 1998.

[26] G. Mastroianni and G. V. Milovanović. *Interpolation Processes: Basic Theory and Applications*. Springer, 2008.

[27] W. McCune. Mace4. URL http://www.cs.unm.edu/~mccune/mace4/.

[28] B. D. McKay and A. Piperno. Practical graph isomorphism, II. *J. Symbolic Computation*, 60:94–112, 2014.

[29] Ž. Mijajlović. *Model Theory*. Novi Sad, 1985.

[30] Ž. Mijajlović. On free boolean vectors. *Publ. Inst. Math*, 64(78):2–8, 1998.

[31] Ž. Mijajlović and A. Pejović. Hopf algebra of projection functions. *Publ. Inst. Math*, 97(111):23–31, 2015.

[32] Ž. Mijajlović, M. Milošević, and A. Perović. Ideal membership in signomial rings. *Univ. Beograd. Publ. Elektrotehn. Fak., Ser. Mat.*, 18:64–67, 2007.

[33] Ž. Mijajlović, D. Doder, and A. Ilić-Stepić. Borel sets and countable models. *Publ. Inst. Math*, 90(104):1–11, 2011.

[34] G. V. Milovanović. A class of orthogonal polynomials on the radial rays in the complex plane. *J. Math. Anal. Appl.*, 206:121–139, 1997.

[35] G. V. Milovanović. Orthogonal polynomials on the radial rays and an electrostatic interpretation of zeros. *Publ. Inst. Math. (Belgrade)*, 64(78):53–68, 1998.

[36] E. B. Saff and R. S. Varga. On incomplete polynomials. In *Numerische Methoden der Approximationstheorie, Band 4*, pages 281–298. Springer, 1978.

[37] T. Sang, F. Bacchus, P. Beame, H. A. Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. In *Proceedings of SAT-04: 7th International Conference on Theory and Applications of Satisfiability Testing*, 2004.

[38] R. Sikorski. *Boolean Algebras*. Springer-Verlag, Berlin, 1969.

[39] T. B. Stojadinović. *Kombinatorne Hopfove algebre*. PhD thesis, Faculty of Mathematics, Belgrade University, 2013.

[40] L. Svenonius. A theorem on permutations in models. *Theoria*, 25:173–178, 1959.

[41] L. G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8:189–201, 1979.

[42] H. Zhang, M. P. Bonacina, and J. Hsiang. Psato: a distributed propositional prover and its application to quasigroup problems. *Jour. of Symbolic Computation*, 21(4-6):543–560, 1996.

Appendix A

Source code listings

A.1 Sudoku solver

```
'''
Sudoku Solver

Usage: sudoku.py <input_file>

@author: Aleksandar
'''
from logic import bool
from logic.core import ast
from logic.abc import p, i, j, k, l
from logic.fo.quantifiers import *


# Sudoku size
n = 3

# cell indices
S = range(n**2)

# cell values
V = {v + 1 for v in S}
Dkl = comb(V, 2)

# we model a Sudoku puzzle in the following way:
# cell(i,j)=k iff p(i,j,k)=1

# there is a value in every cell
f1 = A[i : S].A[j : S].E[k : V](p(i, j, k))
# each value appears in each row
f2 = A[i : S].A[k : V].E[j : S](p(i, j, k))
```

```
# each value appears in each column
f3 = A[j : S].A[k : V].E[i : S](p(i, j, k))

# in one cell there can be only one value
f4 = A[i : S].A[j : S].A[k, l : Dkl](~(p(i, j, k) & p(i, j, l)))


def subsquare(i, j):
    a, b = i // n, j // n
    x = [a * n + c for c in range(n)]
    y = [b * n + c for c in range(n)]
    return prod(x, y)

# each value appears in each subsquare
f5 = bool.true
for sx, sy in prod(range(0, n**2, n), repeat=2):
    f5 &= A[k : V].E[i, j : subsquare(sx, sy)](p(i, j, k))


def set_assumption(i, j, k, assumptions):
    '''
    Fills in a cell(i,j) with k.
    '''
    for ii in S:
        assumptions[p(ii, j, k)] = 0
    for jj in S:
        assumptions[p(i, jj, k)] = 0
    for kk in V:
        assumptions[p(i, j, kk)] = 0
    for ii, jj in subsquare(i, j):
        assumptions[p(ii, jj, k)] = 0
    assumptions[p(i, j, k)] = 1


def load(assumptions):
    '''
    Loads a Sudoku puzzle from an input file into assumptions.
    '''
    import sys
    with open(sys.argv[1]) as f:
        for i, row in enumerate(f):
            for j, k in enumerate(''.join(row.split())):
                if k != '-':
                    set_assumption(i, j, int(k), assumptions)
```

```
def pprint(assumptions):
    '''
    Pretty prints a Sudoku puzzle.
    '''
    sudoku = [['-'] * n**2 for i in range(n**2)]
    for i, j, k in [k.indices for k, v in assumptions.items() if v]:
        sudoku[i][j] = k

    row_sep = '+-{}-+'.format('-+-'.join(['-'.join(['-'] * n)] * n))
    row_tl = '|␣{}␣|'.format('␣|␣'.join(['␣'.join(['{}'] * n)] * n))
    for i, row in enumerate(sudoku):
        if i % n == 0:
            print(row_sep)
        print(row_tl.format(*row))
    print(row_sep)
    print()


# Since we don't have negative literals in sudoku, we only search for
# positive ones.
#
# Negative literal ~p(i, j, k) would mean that we can't put k at (i, j).
# That is only possible if we have already put k somewhere else in i-th
# row, j-th column or (i, j)-th subsquare. But if we had put k somewhere
# else, we have already eliminated all negative literals containing k,
# hence we can't have negative literals.
class Literals(ast.Visitor):

    def __call__(self, term):
        literals = set()
        self.visit((term,), literals)
        return literals

    def default_visit(self, term, literals):
        for t in term:
            if isinstance(t, (bool.algebra.And, bool.algebra.Var)):
                self.visit(t, literals)

    def visit_Var(self, term, literals):
        literals.add(term)

literals = Literals()
```

```
def best_guess(variables):
    '''
    Simple strategy for choosing a cell with least options.
    '''
    sudoku = [[[] for i in range(n**2)] for j in range(n**2)]
    for i, j, k in [var.indices for var in variables]:
        sudoku[i][j].append(k)
    min_i, min_j, unused = min(
        [(i, j, len(vals)) for i, row in enumerate(sudoku)
            for j, vals in enumerate(row) if vals],
        key=itemgetter(2))
    return [(min_i, min_j, k) for k in sudoku[min_i][min_j]]


def ssolve(f, assumptions):
    '''
    Simple backtracking Sudoku solver.
    '''
    f = f.eval(assumptions)
    while literals(f.terms):
        for i, j, k in [literal.indices for literal in literals(f.terms)]:
            set_assumption(i, j, k, assumptions)
        f = f.eval(assumptions)
    if f == bool.Expression(bool.false):
        return None
    elif f == bool.Expression(bool.true):
        return assumptions
    else:
        for i, j, k in best_guess(f.variables):
            assumptions_copy = assumptions.copy()
            set_assumption(i, j, k, assumptions_copy)
            solution = ssolve(f, assumptions_copy)
            if solution:
                return solution

def main():
    assumptions = {}
    load(assumptions)
    pprint(assumptions)

    f = bool.Expression(f1 & f2 & f3 & f4 & f5)
    solution = ssolve(f, assumptions)
    pprint(solution)

if __name__ == '__main__':
    main()
```

A.2 Lattice counter

```
'''
Demonstrates advanced techniques for counting models.

The idea is to decompose a lattice into levels. By going through all
compositions we will create a partitioning of a problem space, as
it can be shown that each lattice can belong to only one composition.

The structure of each composition will allow us to introduce additional
constraints. As a result we will be able to kill more variables, and by
doing so to count lattices on larger domains.

@author: Aleksandar
'''
from logic import bool
from logic.abc import p, i, j, k, l
from logic.fo.quantifiers import *
from logic.cl.runner2 import GPURunner, CPURunner

# increase recursion limit, which is 1000 by default
import sys
sys.setrecursionlimit(5000)


def decompose(s):
    '''Generates all compositions of a given sequence s.'''
    s = tuple(s)
    yield (s,)
    for i in range(1, len(s)):
        for c in decompose(s[i:]):
            yield (s[:i],) + c

def binomial(n, k):
    '''Compute binomial nCk by direct multiplicative method.'''
    if k > n:
        return 0
    if k > n - k: # Take advantage of symmetry of Pascal's triangle
        k = n - k
    acc = 1
    for i in range(1, k + 1):
        acc *= (n - (k - i))
        acc //= i
    return acc
```

```
def assume_reflexive(s, assumptions):
    for i in s:
        assumptions[p(i, i)] = 1

def assume_min_max(s, assumptions):
    '''Fix min and max element.'''
    min_i, *rest, max_i = s
    for i in rest + [max_i]:
        assumptions[p(i, min_i)] = 0
        assumptions[p(min_i, i)] = 1
    for i in [min_i] + rest:
        assumptions[p(max_i, i)] = 0
        assumptions[p(i, max_i)] = 1
    return rest

def assume_unrelated(s, assumptions):
    '''Elements at the same level are unrelated.'''
    for i, j in perm(s, 2):
        assumptions[p(i, j)] = 0

def assume_not_in_relation(s, r, assumptions):
    '''Elements at s level are not smaller than elements at lower levels.'''
    for i, j in prod(s, r):
        assumptions[p(i, j)] = 0


def handle_unary_components(composition, assumptions):
    '''Handles levels with one element.'''
    for c_i, c in enumerate(composition):
        if len(c) == 1:
            i = c[0]
            for next_c in composition[(c_i + 1):]:
                for j in next_c:
                    assumptions[p(i, j)] = 1


# i < j iff p(i,j)=1

n = 9

S = range(n)

assumptions = {}

assume_reflexive(S, assumptions)
```

```
Sr = assume_min_max(S, assumptions)

Dij = comb(Sr, 2)
Dijk= perm(Sr, 3)

# Statement: p is reflexive
#f1 = A[i : S](p(i, i)) # Not needed since we have assumed reflexivity

# Statement: p is antisymmetric
f2 = A[i, j : Dij](~p(i, j) | ~p(j, i))

# Statement: p is transitive
f3 = A[i, j, k : Dijk](~(p(i, j) & p(j, k)) | p(i, k))

# Every two elements have a meet.
f4 = A[i, j : Dij].E[k : S].A[l : S](p(k, i) & p(k, j)
                                     & (~(p(l, i) & p(l, j)) | p(l, k)))

f = bool.Expression(f2 & f3 & f4).eval(assumptions).terms


runner = GPURunner()

n_models = 0
n_c_models = 0

for composition in decompose(Sr):
    c_assumptions = assumptions.copy()
    prev_elements = ()
    coeff = 1
    f_c = bool.true
    for c_i, component in enumerate(composition):
        assume_unrelated(component, c_assumptions)
        assume_not_in_relation(component, prev_elements, c_assumptions)
        coeff *= binomial(len(Sr) - len(prev_elements), len(component))
        prev_elements += component
        if c_i:
            # Each element must be greater than an element at the level below
            f_c &= A[j : component].E[i : composition[c_i - 1]](p(i, j))
    handle_unary_components(composition, c_assumptions)
    expression = bool.Expression(f & f_c).eval(c_assumptions)
    if expression == bool.Expression(bool.true):
        # Reduced to tautology ~ 1
        c_models = 1
    else:
        c_models = runner.count(expression)
```

```
    n_models += coeff * c_models
    n_c_models += c_models

    print(composition)
    print('vars:␣{}'.format(len(expression.variables)))
    print('models:␣{}'.format(c_models))
    print()

print(n * (n - 1) * n_models)
print(n_c_models)
```