UHASSELT
KNOWLEDGE IN ACTION

Maastricht University

# Faculteit Wetenschappen
## *School voor Informatietechnologie*

master in de informatica

***Masterthesis***

***Analyzing the expressiveness of code querying and code mining through logic***

**Jasper Steegmans**

Scriptie ingediend tot het behalen van de graad van master in de informatica

**PROMOTOR :**

Prof. dr. Jan VAN DEN BUSSCHE

**BEGELEIDER :**

Prof. dr. Stijn VANSUMMEREN

**2020**
**2021**

De transnationale Universiteit Limburg is een uniek samenwerkingsverband van twee universiteiten in twee landen: de Universiteit Hasselt en Maastricht University.

UHASSELT
KNOWLEDGE IN ACTION

**www.uhasselt.be**
Universiteit Hasselt
Campus Hasselt:
Martelarenlaan 42 | 3500 Hasselt
Campus Diepenbeek:
Agoralaan Gebouw D | 3590 Diepenbeek



Maastricht University

# Faculteit Wetenschappen
## *School voor Informatietechnologie*

master in de informatica

***Masterthesis***

***Analyzing the expressiveness of code querying and code mining through logic***

**Jasper Steegmans**

Scriptie ingediend tot het behalen van de graad van master in de informatica

**PROMOTOR :**

Prof. dr. Jan VAN DEN BUSSCHE

**BEGELEIDER :**

Prof. dr. Stijn VANSUMMEREN

# Contents

# Nederlandse Samenvatting

In de informatica-industrie worden er gigantische hoeveelheden code geschreven. Zo had Google bijvoorbeeld in 2015 ongeveer twee miljard lijnen aan code over al hun producten heen. Met zulke grote hoeveelheden code, is het moeilijk om een overzicht te hebben en geen enorm grote hoeveelheden gedupliceerde code te hebben. Hiervoor werden er binnen software-engineering verschillende patronen ontwikkeld. Deze patronen zorgen er voor dat de code op zo'n manier gestructureerd wordt dat ze gemakkelijk hergebruikt kon worden. Echter, niet iedere programmeur weet deze patronen goed toe te passen of kent ze volledig niet. Daardoor is het een interessante oefening om door een stuk bestaande code te gaan en te onderzoeken welke patronen er bestaan. Dit noemt men *code patronen minen*. Dit kan dan aantonen dat bepaalde patronen verkeerd toegepast worden of dat er code is die vaak voor komt en die beter in een eigen functie kan gestopt worden.

In code patronen minen beschouwen we een stuk code eigenlijk als data. Dit is in tegenstelling tot het normale geval, waar we code beschouwen als iets wat moet worden uitgevoerd. We zouden ons dan ook de vraag kunnen stellen of we dit niet in een database kunnen doen. Een database er namelijk voor gemaakt om efficiënt vragen te kunnen beantwoorden. Als we de code kunnen opslaan als data in een database is het misschien mogelijk om hier vragen over te stellen. Het is misschien zelfs mogelijk om volledige algoritmes die aan code patronen minen doen te schrijven als een query in zo'n database. Als dat het geval is, zou dat er toe leiden dat er gemakkelijk nieuwe soorten algoritmes die aan code patronen minen doen, kunnen worden bedacht en getest. Ook zou het veel gemakkelijker worden om vragen te stellen over de gevonden patronen om zo enkel de nuttigste patronen terug te geven. Daarom zullen we dus een *querytaal* ontwikkelen over een speciaal soort database en aantonen wat wel en niet mogelijk is met deze query taal een subsets hiervan.

## Eindige Model Theorie

Om de kunnen bewijzen dat een query niet kan worden uitgedrukt in een query taal, kunnen de resultaten van de eindige model theorie gebruikt worden. Aantonen dat iets kan uitgedrukt worden is gemakkelijk, namelijk we geven

simpelweg de formule die de query uitdrukt. Aantonen dat iets niet kan worden uitgedrukt is moeilijker. Typisch moeten we hiervoor aannemen dat iets uitgedrukt kan worden, dan een eigenschap gebruiken en dan leidt de toepassing van die eigenschap tot een contradictie. Uit deze contradictie kunnen we dan afleiden dat onze aanname dat de query kan worden uitgedrukt niet juist is. Maar om tot zo'n contradictie te komen hebben we eerst de eigenschappen nodig. Deze eigenschappen kunnen we vinden in de eindige model theorie voor logica's. Uiteraard moeten we dan onze query talen nog omzetten naar logica's om deze te kunnen gebruiken, maar hier komen we later op terug.

We beginnen in de eindige model theorie met een woordenschat en een structuur over die woordenschat te definiëren. Dit zijn de elementen waarover we een formule zullen evalueren. We introduceren ook eindige structuren en classen van structuren. Een klasse van een structuur is simpelweg een set van structuren die gesloten is onder isomorphismen. Daarna definiëren we een query over een klasse van structuren. Vervolgens introduceren we de verschillende relaties die structuren en klassen kunnen hebben met elkaar. Zo introduceren we bijvoorbeeld isomorphismen, een substructuur zijn van een structuur en een subklasse zijn van een klasse. Een laatste relatie die we introduceren is het partieel isomorfisme tussen twee structuren. Vervolgens definiëren we enkele eigenschappen zoals gesloten zijn onder isomorphismen en bewaard blijven onder isomorphismen. Hierna gaan we over tot de logica's. We definiëren een logica en het concept dat een query *L-definieerbaar* is voor een bepaalde logica L. Vervolgens definiëren we enkele belangrijke logica's, namelijk de propositionele logica, *eerste-orde logica* en tweede-orde logica. We introduceren ook verschillende velden binnen model theorie. Deze velden zijn de klassieke model theorie, onderzoek naar lokale definieerbaarheid en de eindige model theorie. We doen dit omdat alle concepten die we tot nu toe geïntroduceerd hebben allemaal van toepassing zijn op heel de model theorie. We geven ook enkele subsets van de tweede-orde logica, namelijk de universele en de existentiële tweede orde logica en hun monadische tegenhangers.

Vervolgens gaan we over tot het introduceren van het concept van ondefinieerbaarheid en geven we enkele tools die hiervoor gebruikt worden. We geven de compactheidsstelling en de methode van ultraproducten als de eigenschappen waarmee we contradicties kunnen bereiken. Hierna beschrijven we een *Ehrenfeucht-Fraïssé spel* en wat een winnende strategie is voor zo'n game. Vervolgens introduceren we de equivalentie relaties tussen structuren over eenzelfde woordenschat, waarbij de Duplicator een winnende strategie heeft voor het $r$ ronden Ehrenfeucht-Fraïssé spel. We introduceren zo een relatie dus voor iedere $r$. Hierna introduceren we *kwantor rang* voor een formule in de eerste-orde logica. Met deze kwantor rang definiëren we een tweede equivalentie relatie tussen twee structuren. Deze keer zijn de twee structuren equivalent als ze voldoen aan dezelfde set van eerste-orde zinnen. Vervolgens geven we dat deze twee equivalentie relaties equivalent zijn. Hiermee geven we dus een link tussen dat

een query kan worden uitgedrukt in de eerste-orde logica en dat de Duplicator een winnende strategie heeft voor een Ehrenfeucht-Fraïssé spel. Als een formule bestaat van kwantor rang $r$ die een query uitdrukt, dan moeten alle structuren waar de Duplicator een $r$ ronden Ehrenfeucht-Fraïssé spel wint hetzelfde antwoord geven.

Dit laatste resultaat kunnen we dan gebruiken om aan te tonen dat een query niet kan worden uitgedrukt in eerste-orde logica. We doen dit door te bewijzen dat voor iedere $r$ we een set van twee structuren kunnen bedenken waarop de query waar zou moeten zijn in de ene structuur en onwaar in de andere structuur. We tonen dan aan dat de Duplicator dan altijd kan winnen tussen deze twee structuren in $r$ stappen. Dit leidt dan tot een contradictie, aangezien volgens onze equivalentie zouden we hetzelfde resultaat verwachten maar uit de query weten we dat het een verschillend resultaat is. Dit geeft dus aan dat we de query niet in een eerste-orde formule met een kwantor rang $r$ kunnen uitdrukken. Als we dit voor iedere $r$ kunnen bewijzen, dan bewijzen we dat er geen enkele eerste-orde formule is die de query kan uitdrukken.

## Code Patronen Minen

Nu we de benodigde tools hebben besproken kunnen we over gaan tot het bekijken van het domein waar we een query taal voor willen schrijven. In code patronen minen zijn er algoritmes die werken op stringen en substringen. We beginnen met te argumenteren dat een abstracte syntax boom meer geschikt is om patronen te minen. Dit is omdat het ook de relaties tussen de onderdelen van de code bevat en dus meer informatie encodeert dan enkel de karakters van een string. Hierna definiëren we de gelabelde geordende boom en een subboom in zo'n boom. Vervolgens definiëren we een patroon boom, wat een boom is die we kunnen mappen naar een subboom van een echte abstract syntax boom.

Aangezien we altijd patronen minen die gelijkaardig zijn en regelmatig voorkomen, moeten we definiëren wanneer twee patronen gelijkaardig zijn. Een eerste definitie is patroon boom gelijkaardigheid, waarbij twee subbomen eenzelfde patroon boom hebben. Vervolgens definiëren we API set gelijkaardigheid, waarbij we twee stukken code als gelijkaardig beschouwen indien ze dezelfde set van functies aanroepen. Gelijkaardig hieraan definiëren we API sequentie gelijkheid, waar we twee stukken code als gelijkaardig beschouwen indien ze dezelfde set van mogelijke sequenties van functie aanroepen hebben. Tot slot beschouwen we ook nog gelijkaardigheden die komen na een vertaling naar een andere tussenliggende taal. Hierbij vertalen we onze abstract syntax boom naar een abstract syntax boom in een andere taal en kijken dan of de vertaalde patronen gelijkaardig zijn daar.

Vervolgens bekijken we een aantal verschillende code patroon mining algoritmes. We beginnen met het FREQT algoritme dat in het algemeen op bomen

werkt. Dit algoritme zoekt alle subbomen die een minimale frequentie bereiken. Het gebruikt patroon boom gelijkaardigheid en werkt door voor iedere frequente patroon boom, een uitbreiding te maken op diens meest rechtse tak. Daarna wordt er gecontroleerd of deze uitbreiding nog steeds frequent is. We gaan ook in op een aantal optimalisaties van het algoritme. Vervolgens beschouwen we FREQTALS wat een uitbreiding is op het FREQT algoritme dat meer gespecialiseerd is om te werken met abstracte syntax bomen. In FREQTALS voegen we een groot aantal extra beperkingen toe. Deze beperkingen zorgen er voor dat er meer mogelijkheid is tot optimalisaties en dat een menselijke eindgebruiker de resultaten nuttiger vind. Een van de problemen met het FREQT algoritme is namelijk dat het elk paar patronen dat gelijkaardig is terug geeft. Dit is natuurlijk overweldigend voor een menselijke eindgebruiken. Een andere interessante toevoeging van FREQTALS is dat het met de resultaten op de beperkte set resultaten ook nog probeert deze zo hard mogelijk uit te breiden zonder dat het minder vaak voor komt in de abstracte syntax boom.

Hierna introduceren we een aantal mogelijke algoritmen gebaseerd op het gebruik van gelijkaardigheid met een tussenliggende taal. We geven hier twee verschillende aanpakken. In de eerste aanpak vertalen we eerste de abstracte syntax boom en passen we daarna een van de voorgaande algoritmen toe. Dit heeft een aantal problemen, zoals de aanpassing van wat frequente bomen zijn, aangezien meerdere gelijkaardige bomen mogelijks op eenzelfde subboom worden gemapt in de vertaling. Een ander probleem is dat het mogelijk moeilijk is om de gevonden frequente patronen terug te vertalen naar patronen in de originele taal. Dit kan wel mogelijk verholpen worden door extra data in de vertaalde boom bij te houden. Een tweede mogelijke aanpak is om patronen te zoeken in de originele abstracte syntax boom, maar om hun vertaalde boom te gebruiken om te bepalen of ze frequent zijn en wat hun grootte is. Dit leidt dan weer tot zijn eigen set van complicaties aangezien we nu moeten een vertaling doen gedurende het minen. Ook zitten we met het probleem van bomen waarvan er na de vertaling een aantal nodes zijn die niet vertaald zijn. Dit lossen we op door slim te bepalen wanneer een boom altijd losse nodes zal blijven hebben in zijn vertaling en deze bomen niet meer te beschouwen.

## Relationele Meta-Algebra

Nu we code patronen mining algoritmes hebben bekeken, zullen we database query talen bekijken. We doen dit zodat we ons kunnen baseren of laten inspireren door deze talen. Een aantal van de talen die we zullen beschouwen kunnen zelf hun eigen query's queryen, maar niet in het algemeen code.

De eerste taal die we bekijken is de Meta-SQL. In deze taal voegt opgeslagen query's als waarden in een XML kolom toe en de XML transformatie-taal XSLT. De query's worden opgeslagen als een XML encodering van hun abstracte syntax

boom. Vervolgens kunnen we met XML variabelen en de XSLT taal deze XML-documenten aanpassen. Tot slot kunnen we de taal ook evalueren met de 'EVAL' en 'UEVAL' functies. Deze taal bewijst zichzelf enorm krachtig, vooral dankzij de toevoeging van XSLT. Dit heeft tot gevolg dat de taal Turing-compleet is en dat het dus ook niet langer gegarandeerd is dat we een oplossing zullen krijgen voor iedere query. Het voordeel langs de andere kant is dat dit zeer weinig extra werk vergt om te implementeren, aangezien er al onderzocht is hoe XSLT kan geoptimaliseerd worden. Op het moment dat Meta-SQL werd voorgesteld, bestond SQL/XML nog niet. We zouden de Meta-SQL dus kunnen vervangen door SQL/XML met de 'EVAL' en 'UEVAL' functies toegevoegd.

De volgende twee talen die we bespreken zijn de bekende relationele algebra en relationele calculus. We definiëren hiervoor hun operatoren en een database schema waarover ze worden geëvalueerd. We geven aan dat de relationele algebra de basis vormt voor SQL, aangezien ze alle SQL query's kan uitdrukken die enkel de 'SELECT', 'FROM' en 'WHERE' clauses en subquery's gebruiken. Voor de relationele calculus definiëren we ook nog een veilige variant, die equivalent is met de relationele algebra. De relationele calculus is syntactisch beperkt en staat ons dus toe om te bewijzen dat iets niet kan worden uitgedrukt in de relationele algebra. We doen dit door te bewijzen dat iets niet kan in eerste-orde logica, waarvan de veilige relationele calculus een subset is. Aangezien de veilige relationele calculus het dan niet kan uitdrukken en de relationele algebra hiermee equivalent is, kan deze het ook niet uitdrukken.

Vervolgens gaan we over tot de relationele meta-algebra. Dit is een taal gebaseerd op de relationele algebra die operators toevoegt om om te kunnen gaan met opgeslagen query's. Het voegt ook een type systeem toe, waarbij iedere kolom een type heeft dat de ariteit van de opgeslagen query voorstelt. Ook maakt het een distinctie tussen een meta-niveau schema, waar de relaties relationele algebra expressie kunnen bevatten en een object-niveau schema waar dit niet kan. Vervolgens bekijken we hoe de operators van de relationele algebra zijn aangepast, zodat ze logisch en consistent zijn met het nieuwe typesysteem. Daarna bekijken we de herschrijfregels die geïntroduceerd worden om ons toe te staan de relationele algebra expressie aan te passen. Dit gebeurt samen met de 'rewrite' operatoren die voor de waarden in een bepaalde kolom een herschrijfregel een keer of op alle mogelijke plaatsen in de expressie toepast. Vervolgens is er ook de 'extract' operator die uit een bepaalde kolom alle subexpressie van een bepaalde ariteit haalt. Ook voegt de relationele meta-algebra de 'wrap' operator die ons toestaat een constante unaire relatie te maken die enkel een bepaalde data waarde bevat. Vervolgens bespreken we nog de 'eval' operator die de relationele algebra expressie in een bepaalde kolom uitvoert en de resultaten bijvoegt. Tot slot maken we ook de vergelijking met de Meta-SQL, waarin we vaststellen dat de Meta-SQL veel krachtiger is. Ook stellen we vast dat beide querytalen een gelijkaardig doel willen bereiken, maar dat de relationele meta-algebra naar een veel conservatievere route naartoe neemt.

Als laatste querytaal bespreken we de relationele meta-caclulus. Dit is de logische tegenhanger van de relationele meta-algebra en is gebaseerd op de relationele calculus. Ook hier introduceren we een type systeem op de variabelen, analoog aan het systeem in de relationele meta-algebra. Ook voor de herschrijfregels definiëren we een equivalent in de relationele meta-calculus waarbij we toestaan dat er expressie variabelen worden gebruikt in het beschrijven van bomen. Vervolgens vertalen we de verschillende operatoren van de relationele meta-algebra in de relationele meta-calculus met behulp van nieuwe predicaten. Iedere operator heeft een predicaat met variabelen per input waarde en een variabele voor de output waarde. Ook hier definiëren we een veilige variant die equivalent is met de relationele meta-algebra. Deze veilige relationele meta-calculus heeft weer dezelfde voordelen die we hadden bij de veilige relationele calculus.

## Vergelijking tussen aanpakken

Nu we zowel de code patroon mining algoritmes, als de database query talen hebben beschouwd, gaan we de twee aanpakken vergelijken en proberen te verenigen. We zullen dit doen door een serie van query's te beschouwen die typische vragen voorstellen in een systeem waar aan code patroon minen wordt gedaan. We gaan in het begin uit dat een lijst met alle gelijkaardige patronen beschikbaar is.

De eerste query die we zullen beschouwen is een query die alle patronen selecteert die meer dan een vast aantal keren voorkomt. Deze query is dezelfde vraag die FREQT stelt voor een gegeven abstract syntax boom. We kunnen van deze query gemakkelijk aantonen dat deze uitdrukbaar is in de relationele calculus en dus ook de relationele algebra. Dit steunt echter wel op onze assumptie dat er een relatie beschikbaar is die ons alle gelijkaardige paren geeft.

Vervolgens gaan we over naar een query waarbij we over paren van patronen lopen en alle patronen willen waarbij het eerste patronen meer frequent is dan het tweede. Hier kunnen we dankzij een Ehrenfeucht-Fraïssé bewijzen dat dit niet kan worden uitgedrukt in de relationele calculus en dus de relationele algebra. In een poging dit probleem op te lossen introduceren we onze eigen algebra: de *relationele boom-algabra*. In deze algebra voegen we een nieuw soort kolom toe die een boom kan bevatten en voegen we een aantal nieuwe operatoren toe. Deze operatoren zijn de 'match' operator, die controleert dat de wortel knop een bepaald label heeft, en de 'extract' operator die het $n$-de kind van de wortel knoop geeft. We voegen nog een derde operator toe, de 'construct' operator, die ons toestaat om een nieuwe boom met een aantal kinderen van de huidige relatie te maken, met een nieuwe wortel knoop met een gegeven label. We werken ook varianten uit met een typesysteem, naar analogie met de relationele meta-algebra. We werken twee zo'n algebra's uit: een eerste gebaseerd op het aantal kinderen onder de wortel knoop en een tweede gebaseerd op de regels van

een context-vrije grammatica. We tonen ook aan dat deze getypeerde varianten onder bepaalde beperkingen equivalent zijn.

Vervolgens definiëren we ook nog een logische tegenhanger van de relationele boom-algebra: de *relationele boom-calculus*. Hier gebruiken we opnieuw een set van proposities om iedere operator voor te stellen, waarbij de variabelen de input variabelen van de operator en de output zijn. We staan ook toe dat de existentiële en universele kwantor variabelen bevatten die bomen voorstellen. Ook breiden we equivalentie relatie uit zodat deze kan controleren dat bomen equivalent zijn. Hierna tonen we aan dat de relationele boom calculus nog altijd niet sterk genoeg is om onze tweede query uit te drukken.

Om te proberen de tweede query uit te drukken, voegen we nog een operator toe. Deze operator is de 'subtree' operator, waarbij we rijen filteren zodat de voor de overgebleven rijen de bomen van een gegeven kolom een subboom zijn van de bomen van een andere gegeven kolom. Deze uitbreiding noemen we de *relationele subboom-algebra*. We definiëren voor deze algebra ook een nieuwe calculus, waarbij we en predicaat toevoegen dat waar is als de boom in de eerste variabele een subboom is van de boom in de tweede variabele. We noemen deze nieuwe calculus de *relationele subboom-calculus*.

Vervolgens tonen we aan dat de relationele subboom-calculus inderdaad sterk genoeg is om onze tweede query uit te drukken. We doen dit door een aantal vertalingen te definiëren die ons toestaan om tweede-orde logica te gebruiken. Vervolgens definiëren we in tweede-orde logica de formule voor onze tweede query. De geschreven vertalingen zijn zo krachtig dat we zo ook voor andere bewijzen kunnen gebruiken.

Tot dit punt hebben we de assumptie gemaakt dat er een relatie beschikbaar was die alle paren van gelijkaardige patronen bevatte. Uiteraard is dit een heel grote assumptie, aangezien een enorm groot deel van de tijd van een code patroon mining algoritme besteed wordt aan het vinden van deze paren. Het zou dus handig zijn als we deze ook konden uitdrukken in onze query taal. We bewijzen dat we ook gelijkaardigheid kunnen schrijven als een formule in de relationele subboom calculus. Hier steunen we, net zoals bij de tweede query, hard op de kracht van de 'subtree' operator. We lossen dit bewijs op door paden te definiëren die van de wortel van de boom vertrekt en naar de twee gelijkaardige patronen loopt. De reden dat we dit moeten doen is dat we niet rechtstreeks uit de 'subtree' operator kunnen bepalen wat de locatie is van de subboom in de grotere boom. Hiermee bewijzen we dus dat patroon boom gelijkaardigheid kan worden uitgedrukt in de relationele subboom-algebra. Dit laat ons dan toe om een intuïtieve beschrijving te geven van hoe FREQT en FREQTALS zouden kunnen worden uitgedrukt in de relationele subboom-algebra.

Na de patroon boom gelijkaardigheid te hebben uitgedrukt, kunnen we ons ook afvragen of we andere soorten gelijkaardigheid kunnen uitdrukken. Voor de API set en API sequentie gelijkaardigheid geven we twee intuïtieve manieren om het probleem aan te pakken, waarvan we denken dat het tot een oplossing zal leiden. Voor de gelijkaardigheden gebaseerd op een tussenliggende taal concluderen we dat we te weinig uitdrukkingskracht hebben. Hiervoor zouden we een tegenhanger moeten hebben van de 'rewrite' operatoren van de relationele meta-calculus. Het is dus zeer waarschijnlijk dat dit niet uit te drukken is in de relationele subboom algebra.

Tot slot vergelijken we de relationele subboom algebra met de relationele meta-calculus en algebra en met Meta-SQL. Voor de relationele algebra concluderen we dat, ondanks dat ze op elkaar lijken, ze elkaars operatoren niet kunnen uitdrukken. De relationele meta-algebra heeft bijvoorbeeld de 'rewrite' operatoren, die waarschijnlijk niet kunnen worden uitgedrukt in de relationele subboom algebra. Bovendien verliezen we in de relationele meta-algebra het concept van ariteit van een relationele algebra expressie. Voor Meta-SQL blijft de conclusie gelijkaardig aan de vergelijking tussen de relationele algebra en Meta-SQL. Ook hier zijn ze sterk gebaseerd op hetzelfde idee, aangezien ze alle twee hun query's opslaan in een boomstructuur. Opnieuw is hier XSLT veel krachtiger dan de relationele subboom-algebra. Ook heeft de relationele subboom-algebra geen 'eval' operator, aangezien het ontworpen is om code te queryen en niet relationele algebra expressies.

# Chapter 1

# Introduction

In the IT industry developers write many lines of code. As an example, the single repository Google used in 2015 contained about two billion lines of code [PL16]. It is not unlikely that other tech giants like Facebook, Amazon, Apple and IBM have a similar amount of lines of code, if not more. As a point reference for how large some other open source projects are, consider the Linux kernel. The Linux kernel is one of the largest open source projects and in March 2021 contained about 28.8 million lines of code, 21.3 million of which were actual code [Lar21]. Between Linux minor versions there is about two months and for version 5.12 there were about 508 000 lines of code added and 312 000 removed. Writing such a vast quantity of code leaves is with a massive treasure trove of data to see how people program.

## 1.1 Code Pattern Mining

In software engineering many patterns and techniques have been developed to help programmers structure their code so that it can be reused better. These typically operate at the level of larger blocks of code like classes and methods and how these interact with each other. Examples of such patters and the Observer pattern, where one class is defined an observable and then many other observers can observe the observable. All these observers then get notified when the state of the observable changes. Here we define what functions two sets of classes must implement for this to work. However, there are also patterns at a more detailed level, such as at the code within the function. As an example consider an array where we call a series of map and filters one after the other on it. This might be a common operation that returns often in the code and thus form a pattern at the level of the lines of code being executed.

However, not all patterns are created equal and are equally liked. Some recurring patterns in code might be the use of the class patterns. For example, assume we call a lot of similar functions of some 'Factory' class, which is a type of

class in the 'Builder' pattern. This means we are using the pattern as intended. However, if we find a lot of duplicate code where only the variable names are different, this might be an indication that this code need to be refactored into its own function which is passed these variables. On top of this, programmers will often look at previously written code and copy this code and modify it for their own use cases. This can then lead to bad code patterns propagating through the codebase without people being aware of them.

On top of this inequality of patterns, not every codebase has people who define the patterns to be used beforehand and then implement these. As much as the provided patterns help structure the code and make it more reusable, not everyone is aware of the patterns or knows how to apply them. On top of this, if a codebase becomes rather large, it can be difficult to determine what patterns have been used after the fact. The reason a programmer might want to do this, is to refactor their code. Say they recently learned about a lot of the software engineering patterns and want to apply them to their own codebase. Before they can start applying these patterns, it would be useful to know what patterns they have unintentionally already been using. This would allow them to have an overview of what there already is, so they can consider whether these patterns are applied correctly and where new patterns need to be introduced.

All of these problems could be solved with *code pattern mining*. In code pattern mining, we give an algorithm a codebase and ask it to find all the code patterns that occur frequently within it. However, if we simply returned every frequent pattern, an end-user would be overwhelmed. Thus, we do some filtering of the output in an attempt to display only the most interesting patterns. Of course, what is considered interesting and what patterns should be considered similar varies. As a result, there are many code pattern mining algorithms available. However, using these algorithms we can address the problems mentioned earlier. For example, we could mine for patterns to determine what patterns already exist in a codebase to allow us to introduce new patterns. We could also use the mined patterns to determine what should be refactored into its own class or function.

## 1.2 Query Languages

So far we have been describing algorithms to search through large amounts of data. However, in databases we already have languages designed to specifically to efficiently query large amount of data stored in a database. Queries in these languages are automatically optimized, so that the developer creating them does not have to worry about optimizing their code. Since in the previous section, we have already considered code to be data that we can search through, why not consider storing it in a database and querying it that way. After all, most data can be represented in a relational database in some way.

As a result of this logic, there have been different approaches to storing queries in a special type of relational database. These approaches then also add functionality to the query languages, so they can handle the new type of data. Typically, they add functions to search through and modify the stored queries. On top of this, they typically also add an evaluation function which allows them to execute the stored and modified queries.

An example of such a query language is *Meta-SQL*. Here we add the XML datatype and XSLT to SQL to allow us to store and manipulate the stored queries. We also add an 'EVAL' and an 'UEVAL' function to allow us to execute these modified queries. Another such language is the *relational meta-algebra*. This is an extension of the relational algebra, that forms the foundation of SQL. In the relational meta-algebra we add functionality to the relational algebra in the form of 'extract', and 'rewrite' operators. We also add a new column type to store these relational algebra expressions. Finally, the relational algebra also adds an 'eval' operator, which allows us to execute the stored and modified queries.

Knowing that these query languages exist, the question arises whether they can also do code pattern mining. This is in fact the question that this thesis attempts to answer; can we define a query language to do code pattern mining? To prove that a language can do something we simply need to give an algorithm for it. However, to prove that a language cannot express something we need additional theorems that we can use. Typically, we use these theorems and assume something can be expressed and then with the theorem arrive at a contradiction, proving that our assumption was wrong. For logical languages, such as *first-order logic*, these theorems can be found in *finite model theory*.

We shall start by first introducing finite model theory and its basic concepts in Chapter 2. After this we shall introduce the tool that we cannot express certain things in first-order logic: Ehrenfeucht-Fraïssé games. In Chapter 3 we shall introduce code pattern mining. We do this by first giving some basic concepts that are used in the similarities and the algorithms. Then we introduce the similarities and finally the algorithms themselves. After this, in the Chapter 4 we introduce and discuss the different query languages. We first discuss Meta-SQL, followed by the relational algebra and its logical counterpart the relational calculus. After these we discuss the relational meta-algebra and its logical counterpart the relational meta-calculus. We compare between these different query languages and also discuss important subsets of them. Finally, in Chapter 5 we finally bring all the pieces of the previous chapters together and answer the question we set out to answer. We do this by proposing two queries and checking what query languages can express them. When we cannot express one of them, we expand the relational algebra with new operators checking which operators exactly allow us to express this query. Finally, we also come back to the similarities and discuss how we could implement some of

them. We then also use this to discuss how we would implement entire code pattern mining algorithms in our query language. Finally, we compare the query language we have defined with the other query languages we discussed earlier.

# Chapter 2

# Finite Model Theory

This thesis will discuss quite a disparate set of methods to query code. As such, a considerable amount of time was spent in understanding *finite model theory*. This is because finite model theory is the field devoted to understanding and comparing the expressiveness of logics. This can then typically be used to check the expressiveness of a logic that is equivalent to a query language. On top of that it also provides tools to prove that certain queries cannot be expressed, showing what the limits are of these languages. Initially we will discuss the basics of finite model theory, such as structures, collections of structures and some commonly used properties about them. Next we will dive into more detail with Ehrenfeucht-Fraïssé games, a useful tool for determining that something cannot be expressed. We will explain what this game is, how it relates to logics and how it can be used to prove a query cannot be expressed in a given logic.

## 2.1 Basics

### 2.1.1 Building blocks

To be able to define anything that could be considered a model, we must first define what we are defining our model over. This is called the vocabulary.

**Definition 2.1.1.** A *vocabulary* is $\sigma = \{R_1, \ldots, R_m, c_1, \ldots, c_s\}$, where $R_1, \ldots, R_m$ is a set of relation symbols each with their own arity and $c_1, \ldots, c_s$ a set of constant symbols.

The vocabulary defines the set of relation symbols, their arities and a set of constants. A vocabulary is similar to a schema in a database: it simply defines the structure of the database but does not specify the actual data that the database instance with the given schema will be storing. For example, if we wish to describe a directed graph with some special node in it, we would define some vocabulary with a relation symbol $N$, which is a unary symbol and contains all the nodes. We would also define a relation symbol $E$ which

will stand for a binary relation which contains all the edges between pairs of nodes. Lastly we would introduce some constant $s$, which will represent some special node. Note how we have only defined what these relation symbols will stand for but not their actual values. This would mean we'd have a vocabulary $\sigma = \{N, E, s\}$. The reason for adding some special node $s$ is that it allows us to always have a certain reference node, no matter the instance. Other examples of constants are 0 and 1 in arithmetic, where they are the neutral element for the $+$ and $*$ operators respectively.

Now that we have a vocabulary we can actually start defining a structure over the vocabulary. Thus, when we have a structure over a vocabulary $\sigma$, it is called a *$\sigma$-structure*. A structure is created over a given *universe* which contains all the elements that can be used to create the structure. If this universe is a finite universe, we get a *finite structure*.

**Definition 2.1.2.** Given a vocabulary $\sigma$, a *$\sigma$-structure* $\boldsymbol{A} = (A, R_1^{\boldsymbol{A}}, \ldots, R_m^{\boldsymbol{A}}, c_1^{\boldsymbol{A}}, \ldots, c_s^{\boldsymbol{A}})$ is a tuple where $A$ is a set called its *universe*. Each $R_i^{\boldsymbol{A}}$ with $1 \leq i \leq m$ is a relation on $A$ such that $\text{arity}(R_i^{\boldsymbol{A}}) = \text{arity}(R_i)$, mapping each $R_i$ from $\sigma$. Each $c_j^{\boldsymbol{A}}$ with $1 \leq j \leq s$ is a distinguished element of $A$ mapping each $c_j$ from $\sigma$ to an actual value.

**Definition 2.1.3.** A *finite structure* is a $\sigma$-structure with a finite universe.

Creating a structure over a given universe is similar to creating a database instance over a known schema and the universe is the set of all values you could give to any column. This would mean that if a relation has an integer column and a different column which can store characters the universe would both contain all integer numbers and all characters. A structure then is an assignment of actual relations, which is a set of tuples, to the relation symbols. This would be akin to creating a database instance over a given universe. If this universe is a finite universe, we get a *finite structure*. This would be similar to having a database instance where the values come from a limited universe, such as all integers that can be represented in a 32-bit number. This is exactly why the study of finite structures, also known as *finite model theory* is so closely tied to databases: we commonly work with only a finite universe in databases.

For our example we will create the relations shown in Table 2.1 and assign $s^{\boldsymbol{A}}$ the value of 1 to create a structure $\boldsymbol{A} = (A, N^{\boldsymbol{A}}, E^{\boldsymbol{A}}, s^{\boldsymbol{A}})$ that represents the graph in Figure 2.1. This shows that the universe $A$ has to at least be $\{1, 2, 3, 4, 5, 6, 7\}$, but it could just as easily be the set of all natural numbers $\mathbb{N}$, or the set of all natural numbers from 1 to 100. For our example we will assume $A = \{1, 2, 3, 4, 5, 6, 7\}$ which means that the example is also an example of a finite structure. If we had chosen $\mathbb{N}$, then we would have had an infinite structure.

Now that a single structure has been defined, it makes sense to consider a group of structures. This is called a *class of structures*. To define this how-

| $N^{\boldsymbol{A}}$ |
|---|
| node |
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |
| 7 |

| $E^{\boldsymbol{A}}$ | |
|---|---|
| start | end |
| 1 | 2 |
| 2 | 1 |
| 2 | 3 |
| 3 | 4 |
| 4 | 1 |
| 5 | 4 |
| 5 | 6 |

Table 2.1: The relations of an example structure

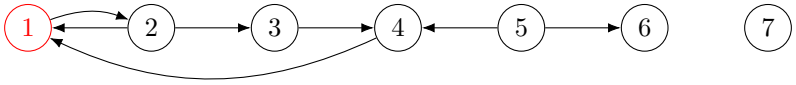



Figure 2.1: A visual representation of the graph represented in the example structure, with the node represented by $s$ in red

ever, we require *isomorphism*, which will be explained in more detail in Definition 2.1.6. However, to intuitively understand it, it suffices to know that when two structures are isomorphic to each other we have a mapping of the elements of the universe one structure to the elements of the universe of the other structure and when applied to one structure it gives the other structure.

**Definition 2.1.4.** A *class of $\sigma$-structures* is a collection $\mathcal{C}$ of $\sigma$-structures that is closed under isomorphisms, which means that if $\boldsymbol{A} \in \mathcal{C}$ and $\boldsymbol{A}$ is isomorphic to $\boldsymbol{B}$ then $\boldsymbol{B} \in \mathcal{C}$.

For our example this would mean that a class $\mathcal{C}$ of every structure isomorphic to $\boldsymbol{A}$ would be an infinite set of structures where for each structure the universe is a set of 7 items which can be mapped to the universe $A$ and when this mapping is applied to the relations of this structure, it results in the structure $\boldsymbol{A}$.

Considering we have a class of structures we might want to extract certain features from these classes or check if certain properties hold on certain classes. This is achieved through what is called a *query* with a given arity.

**Definition 2.1.5.** Given a class $\mathcal{C}$ and $k$ a positive integer, a $k$-ary *query* on $\mathcal{C}$, is a mapping $Q$ with domain $\mathcal{C}$ such that $Q(\boldsymbol{A})$ is a $k$-ary relation for each $\boldsymbol{A} \in \mathcal{C}$ and such that $Q$ is preserved under isomorphisms, which means that if $h : \boldsymbol{A} \to \boldsymbol{B}$ is an isomorphism with $\boldsymbol{A}, \boldsymbol{B} \in \mathcal{C}$ then $Q(\boldsymbol{B}) = h(Q(\boldsymbol{A}))$.

A query can return a range of data types, from a Boolean indicating whether a given graph is a fully connected graph, to the set of all pairs of nodes from a given graph that are connected by a path of less than 5. If our query is Boolean

we can also use it to identify some *subclass* of the given class by considering a class of all structures where the query evaluates to either true or false. This would of course still remain a class since queries are preserved under isomorphisms. In Section 2.1.3 we go into more detail why being preserved under isomorphism is a useful property.

For our example, a query that could be asked is REACHABLE, which is a unary query described as

$$R(\boldsymbol{G}) = \{a \in N \mid \exists p \text{ a path from } s \text{ to } a\}$$

for some structure $\boldsymbol{G}$ over our vocabulary $\sigma$. This query executed over our example structure $\boldsymbol{A}$, results in $\{1, 2, 3, 4\}$. Note that we could have also written our query to return a set of unary tuples instead of a set of elements. Typically, in a unary query a set of elements is returned rather than a set of tuples. On queries of a higher arity however, the query always returns a set of tuples.

### 2.1.2 Relationships

Now that the basic building blocks of model theory have been laid out, we can define relationships between these building blocks. Earlier for Definition 2.1.4 we already intuitively explained the concept of isomorphism.

**Definition 2.1.6.** Given a vocabulary $\sigma$, two $\sigma$-structure $\boldsymbol{A} = (A, R_1^{\boldsymbol{A}}, \ldots, R_m^{\boldsymbol{A}}, c_1^{\boldsymbol{A}}, \ldots, c_s^{\boldsymbol{A}})$ and $\boldsymbol{B} = (B, R_1^{\boldsymbol{B}}, \ldots, R_m^{\boldsymbol{B}}, c_1^{\boldsymbol{B}}, \ldots, c_s^{\boldsymbol{B}})$ an *isomorphism* between $\boldsymbol{A}$ and $\boldsymbol{B}$ is a mapping $h : A \to B$ that

- $h$ is a bijective function, which means it is an injective (one-to-one) and surjective (onto) function
- for every constant symbol $c_j \in \sigma$ with $1 \leq j \leq s$, $h(c_j^{\boldsymbol{A}}) = c_j^{\boldsymbol{B}}$ holds
- for every relation symbol $R_i \in \sigma$ with an arity $t$, $1 \leq i \leq m$ and for every $t$-tuple $(a_1, \ldots, a_t)$ over $A$ we have that $R_i^{\boldsymbol{A}}(a_1, \ldots, a_t)$ if and only if $R_i^{\boldsymbol{B}}(h(a_1), \ldots, h(a_t))$

If there exists an isomorphism between $\boldsymbol{A}$ and $\boldsymbol{B}$, we can also say that $\boldsymbol{A}$ and $\boldsymbol{B}$ are *isomorphic*.

Note that because an isomorphism is bijective, it is also invertible, which means that finding an isomorphism from a structure to a different one also means finding an isomorphism the other way around. This is also why we often refer to an isomorphism between two structures, rather than strictly from one structure to another. An isomorphism is a formal version of saying that two structures express the same relationships and constants if we look at them differently. This concept is very useful, since it allows us to easily and formally state that something must hold for all structures that are similar in some regard, like the class of all structures that represent a fully connected graph, or that an operation applied to similar structures should have similar outcomes, as is the case for queries.

| $N^{\boldsymbol{B}}$ |
|---|
| node |
| a |
| b |
| c |
| d |
| e |
| f |
| g |

| $E^{\boldsymbol{B}}$ | |
|---|---|
| start | end |
| a | b |
| b | a |
| b | c |
| c | d |
| d | a |
| e | d |
| e | f |

Table 2.2: The relations of a structure isomorphic to the running example

For our example a structure $\boldsymbol{B} = (B, N^{\boldsymbol{B}}, E^{\boldsymbol{B}}, s^{\boldsymbol{B}})$ isomorphic to the running example structure $\boldsymbol{A}$ is given in Table 2.2. Assume that $B = \{a, b, c, d, e, f\}$ and that $s^{\boldsymbol{B}} = a$. To show that $\boldsymbol{B}$ is isomorphic to $\boldsymbol{A}$, we must show that there is a mapping between their universes $A$ and $B$ for which the three requirements listed in Definition 2.1.6 hold. This mapping $h$ is very easy to find in this case: $h(a) = 1, h(b) = 2, \ldots, h(g) = 7$. This function is clearly bijective, $h(s^{\boldsymbol{B}}) = h(a) = 1 = s^{\boldsymbol{A}}$ and an element-wise application of $h$ on $N^{\boldsymbol{B}}$ and $E^{\boldsymbol{B}}$ clearly results in $N^{\boldsymbol{A}}$ and $E^{\boldsymbol{A}}$.respectively.

Now that everything required for a structure has been formally defined, we will define some common relations on structures, starting with the 'subset relations' for both structures and classes of structures. For structures this relation is called a *substructure* and for classes of structures this is called a *subclass*. Note that a method of construction for a subclass was already given earlier for Definition 2.1.5.

**Definition 2.1.7.** Given a vocabulary $\sigma$, two $\sigma$-structure $\boldsymbol{A} = (A, R_1^{\boldsymbol{A}}, \ldots, R_m^{\boldsymbol{A}}, c_1^{\boldsymbol{A}}, \ldots, c_s^{\boldsymbol{A}})$ and $\boldsymbol{B} = (B, R_1^{\boldsymbol{B}}, \ldots, R_m^{\boldsymbol{B}}, c_1^{\boldsymbol{B}}, \ldots, c_s^{\boldsymbol{B}})$, $\boldsymbol{B}$ is a *substructure* of $\boldsymbol{A}$ if $B \subseteq A$ and for every $R_i^{\boldsymbol{B}}$ $1 \leq i \leq m$, $R_i^{\boldsymbol{B}} = R_i^{\boldsymbol{A}} \cap B^t$ with $t$ the arity of $R_i$ holds and for every $c_j^{\boldsymbol{B}}$ $1 \leq j \leq s$, $c_j^{\boldsymbol{B}} = c_j^{\boldsymbol{A}}$ holds. $\boldsymbol{B}$ is called a *substructure of* $\boldsymbol{A}$ *generated by* $D$, written as $\boldsymbol{B} = \boldsymbol{A} \restriction D$, if it is a substructure of $\boldsymbol{A}$ and its universe is $B = D \cup \{c_1^{\boldsymbol{A}}, \ldots, c_s^{\boldsymbol{A}}\}$

**Definition 2.1.8.** Given a vocabulary $\sigma$, two classes of $\sigma$-structures $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{B}$ is a *subclass* of $\boldsymbol{A}$ if $\mathcal{B} \subseteq \mathcal{A}$. That is to say that every structure in $\mathcal{B}$ must also appear in $\mathcal{A}$.

Since a subclass is still a class of structures, this means that its structures still need to be closed under isomorphism, which means that not every subset of a class of structures is also a subclass of that same class. As for substructures, this can simply be seen as taking the universe of the structure, removing some elements of it that are not assigned to constants and then removing all tuples that contain elements that were removed from the universe. The resulting structure is then a substructure of the original structure. In databases this

| $N^{\boldsymbol{B}}$ |
|---|
| node |
| 1 |
| 2 |
| 3 |
| 5 |

| $E^{\boldsymbol{B}}$ | |
|---|---|
| start | end |
| 1 | 2 |
| 2 | 1 |
| 2 | 3 |

Table 2.3: The relations of an example structure

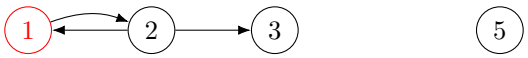



Figure 2.2: A visual representation of the graph of $\boldsymbol{B} = \boldsymbol{A} \restriction \{1, 2, 3, 5\}$, with the node represented by $s$ in red

would mean limiting the values that are allowed to appear in columns and then removing those rows that contain now invalid values.

For our running example the only subclass we could take are the class $\mathcal{C}$ itself, since clearly it is a class and every structure in it is also in $\mathcal{C}$, and the empty class $\emptyset$ since every structure in it is also part of the class $\mathcal{C}$ and the empty class is clearly a class, since for every structure in it, its isomorphism is also part of the class. As for substructures, an example substructure is given by Table 2.3 which is $\boldsymbol{B} = \boldsymbol{A} \restriction \{1, 2, 3, 5\}$. The graph for this is also shown in Figure 2.2.

Finally, we will define *partial isomorphism* which is a weaker form of isomorphism, but still allows us to express a certain degree of similarity between two structures.

**Definition 2.1.9.** Given a vocabulary $\sigma$, two $\sigma$-structures $\boldsymbol{A}$ and $\boldsymbol{B}$, a *partial isomorphism* between $\boldsymbol{A}$ and $\boldsymbol{B}$ is an isomorphism between a substructure of $\boldsymbol{A}$ and a substructure of $\boldsymbol{B}$.

This means that a structure is always partially isomorphic to its substructures, since you can trivially find a substructure of it that is isomorphic to the given substructure: that substructure itself. Partial isomorphism is particularly useful when trying to make a statement about two structures that have a differently sized universe.

For our running example, there will be two sets of $\sigma$-structures that are partially isomorphic to each other: there is a set of graphs where the special node $s$ has no edge that goes from $s$ to itself, often called a self-loop, and a group where $s$ does have a self-loop. This is because the smallest substructure that can be taken, is the one where the only element in the graph is the node given by $s$. At this small a structure there are clearly only two possible graphs: with and

without the self-loop. Clearly these two are not isomorphic to each other, but this the structures within each set are clearly partially isomorphic to each other: map the one element in the universe to the one element in the other structures universe and that is an isomorphism for the substructure. There is technically also a third set, which would be a substructure generated by all graphs with some universe and empty relations for relation symbols $N$ and $E$. However, this third set does not make much sense to consider, since $s$ is supposed to represent some node, but since $N$ is empty it is just some element from the universe. Finally, it is also important to say over what universe two things are partially isomorphic: if they are partially isomorphic over the substructure generated by a universe of a single element, this clearly is a lot weaker than if it were partially isomorphic over the universe of one of the structures.

### 2.1.3 Properties

Now that both the building blocks and relationships have been discussed, we will formalize some properties about them that were already used. These are useful to allow us to express complicated concepts with a single word and allow us to reason about the behavior of more easily. Consider the property of *being closed under isomorphism* and the closely related property of *being preserved under isomorphism*, which were mentioned in Definition 2.1.4 and Definition 2.1.5 respectively.

**Definition 2.1.10.** A set of structures $\mathcal{C}$ is called *closed under isomorphism* if for every structure $\boldsymbol{A} \in \mathcal{C}$, all structures that are isomorphic to $\boldsymbol{A}$ are also part of $\mathcal{C}$.

**Definition 2.1.11.** Given $k$ a positive integer, assume $f : \mathcal{C} \to X^k$ is a function from some set of structures $\mathcal{C}$ to $X^k$, where $X$ is the union of all universes of all structures in $\mathcal{C}$. On top of this $f$ has the restriction that when it is applied to $\boldsymbol{A} \in \mathcal{C}$ with universe $A$, it can only produce a result from $A^k$. $f$ is said to be *preserved under isomorphism* if for every isomorphism $h : \boldsymbol{A} \to \boldsymbol{B}$ with $\boldsymbol{A}$ and $\boldsymbol{B}$ structures, $f(\boldsymbol{B}) = h(f(\boldsymbol{A}))$ and $h^{-1}(f(\boldsymbol{B}) = f(\boldsymbol{A})$.

Being closed under isomorphism is a useful property for any set of structures, since it makes functions or queries over it that are preserved under isomorphism make more sense. Consider for example, if we had a set of structures that was not closed under isomorphism, and we applied a query to it. This would mean that there is a structure that is not part of the set and is isomorphic to a structure that is part of the set. We would therefor be able to easily calculate the result of a query on the structure that is not part of the set. We can do this by using its isomorphic counterpart in the set and the isomorphism between the structures. This means we could calculate the result for the query without having to know the structure itself, but for some reason we choose to not include this structure in the set we are querying. This usually does not make much sense. That is why, when working with functions that are preserved under isomorphism, we normally also work with classes that are closed under isomorphism.

### 2.1.4 Logics

Earlier we defined queries simply as a function that maps a structure from a class to some $k$-ary relation. This definition is purely semantic. In practice however, queries need to be defined in some programming language. To solve this it is often considered whether a query can be written or *defined* within a certain language called a *logic*. Often logics will have clear syntactic restriction, making it easy to determine what is and is not an expression in that logic.

**Definition 2.1.12.** Given a logic $L$, a class of structures $\mathcal{C}$ and a $k$-ary query $Q$ on $\mathcal{C}$, $Q$ is *$L$-definable* if there is an $L$-formula $\varphi(x_1, \ldots, x_k)$ with $x_1, \ldots, x_k$ such that $Q(\boldsymbol{A}) = \{(x_1, \ldots, x_k) \in A^k \mid \boldsymbol{A} \models \varphi(x_1, \ldots, x_k)\}$. Here $\boldsymbol{A} \models \varphi(x_1, \ldots, x_k)$ is true if a given assignment of $x_1, \ldots, x_k$ makes the formula $\varphi(x_1, \ldots, x_k)$ evaluate to true in the structure $\boldsymbol{A}$. If $Q$ is a Boolean query, then it is *$L$-definable* if there is an $L$-sentence $\psi$ such that $Q(\boldsymbol{A}) = \mathit{true} \Leftrightarrow \boldsymbol{A} \models \psi$. A sentence is simply a formula without free variables. $L(\mathcal{C})$ denotes the collection of all $L$-definable queries on $\mathcal{C}$.

What has been described, is also known as *uniform definability*. Uniform definability means that the same $L$-formula is used for the query on each structure in the class the query is defined on. This concept is similar to the requirement that regardless of the input values, the same algorithm should always produce the outcome. This also means that is if the query is defined on a class, that same formula will work on any subclass of that class.

Definability now allows us to express how powerful a language is, by stating what queries it can and cannot define on specific classes of structures. This is one of the most common uses of model theory and its tools: allowing the comparison of very different languages by showing what they can and cannot express. This also then has implications on the complexity of solving problems written in these languages, typically allowing to set upper bounds on it. However, this question of definability is always tied to a specific class and as such there are three special cases that are often considered and considered their own subdomain within the larger domain of *model theory*:

- $\mathcal{C}$ is the class $\mathcal{S}$ of all structures, both finite and infinite, over a given vocabulary. The uniform definability of formulas of a particular logic on this class is mainly studied within *classical model theory*.
- $\mathcal{C}$ consists of a single infinite structure and all structures isomorphic to it. This then gives rise to *local definability*, which is called 'local' because only a single structure should be considered, since queries are closed under isomorphisms. Examples of such structures are the structure of arithmetic over the natural numbers $\boldsymbol{N} = (\mathbb{N}, +, \times)$, and its counterpart for the real numbers $\boldsymbol{R} = (\mathbb{R}, +, \times)$
- $\mathcal{C}$ is the class $\mathcal{F}$ of all finite structures over a given vocabulary. In this case uniform definability of formulas of logics on this class is called *finite model theory*.

**Propositional logic** Now that it has been explained what definability is and why it is useful, let us consider a couple of common logics whose definabilities are studied. Firstly there is *proposition logic*, where we can write sentences over a given set of propositions, typically $\{p, q, r, \dots\}$ or $\{p_1, p_2, \dots\}$ which can be either true or false, and connect them with connectives, typically $\{\neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$. This logic is typically not studied within model theory, however it is often useful in introducing basic concepts, such as the idea of proofs and logical consequences. It is also useful in that it provides a simpler environment to prove and understand the compactness theorem and the completeness theorem. On top of this, it can also be seen as a restricted version of first-order logic, if we consider a proposition to be a relation with an arity of zero. This thus means that a proposition will either represent the empty set $\emptyset = \{\}$, which represents false, or the set $\{()\}$, which represents true. It is assumed that the reader understands propositional logic.

**First-order logic** Next up is the logic that will get most of the focus due to the complexity required to solve problems within it and its close relation with database languages, like SQL. It also has some well understood tools that can be used to prove definability and non-definability on it. *First-order logic*, often shortened to *FO* logic, is very similar to the propositional logic in that it uses the same connectives which have the same interpretation. However, it is different in that it uses the relations from the vocabulary of the structures the formula is being defined over, rather than propositions, which can be considered relations of arity zero. Because relations now have an arity that is higher than zero, the universe of the structure over which we are evaluating our formula comes int to play.

On top of that, these relations can contain variables and constants, for example $R(x, c)$ where $R$ is a relation with arity 2, $x$ is a variable which could take some value from the universe of the structure and $c$ is a constant from the vocabulary with a value from the universe of the structure assigned by the structure. Two variables $x_1$ and $x_2$ can also be compared to each other using $x_1 = x_2$ and $x_1 \neq x_2 \equiv \neg(x_1 = x_2)$.

Finally, first-order logic adds the concept of quantifiers: $\forall x_1$ and $\exists x_2$ respectively called the "for all" or *universal quantifier* and the "exists" or *existential quantifier*. When quantifying over multiple variables rather than write a quantifier for each variable, we can group them all together. As an example, rather than write $\forall x_1(\forall x_2(\exists x_3(\forall x_4)))$ we can write $\forall x_1, x_2(\exists x_3(\forall x_4))$, this grouping all variables following each other using the same quantifier. If all these quantifiers are applied to the same section of the formula, we can also remove the brackets for them. For example $\forall x_1, x_2(\exists x_3(R(x_1, x_2) \wedge R(x_1, x_3)))$ could be rewritten as $\forall x_1, x_2 \exists x_3(R(x_1, x_2) \wedge R(x_1, x_3))$, but $\forall x_1, x_2((\exists x_3(R(x_1, x_3))) \wedge R(x_1, x_2))$ can not be rewritten in this way. Again the assumption is made that the user is

already familiar with first-order logic and understands what the previous terms mean and how a first-order logic formula is interpreted.

For our running example let us define the binary query 2-PATH, which we would have previously defined as:

$$2P(\boldsymbol{G}) = \{(a, b) \in N^2 \mid \exists p \text{ a path of length 2 from } a \text{ to } b\}$$

In first order logic this would be written as:

$$\varphi_{2P}(a, b) = N(a) \wedge N(b) \wedge (\exists c(E(a, c) \wedge E(c, b)))$$

Note how here we have used $\varphi_{2P}$ instead of directly $2P$. This is not strictly required but sometimes preferred so that when it is used as a subformula in some other FO formula, it cannot be confused for some relation.

**Second-order logic** Finally, there is a class of logic that contains the entirety of first order logic but goes even further: *second-order logic*. Second-order logic adds relation variables, typically denoted as $S_1, S_2, \ldots$, and quantifiers over these variables. This means that statements can also be made about the existence of a relation for which a certain property holds, or that a property holds for all relations. This means that second order logic allows us to say sentences such as"The universe contains an even amount of element" by saying "There exists some relationship of arity two in which each number appears exactly once and only either in the first or second position":

$$\begin{aligned}\exists S \forall x \big( & (\exists y\,(S(x, y) \vee S(y, x))) \wedge (\neg S(x, x)) \wedge \\ & \big(\neg \exists y_1, y_2 \big( \big(y_1 \neq y_2 \wedge ((S(x, y_1) \wedge S(x, y_2)) \vee (S(y_1, x) \wedge S(y_2, x)))\big) \vee \\ & (S(x, y_1) \wedge S(y_2, x))\big)\big)\big)\end{aligned} \tag{2.1}$$

This could not be expressed in first-order logic since it is impossible to make a statement over the existence of some relation that is not part of the structure. The reason this is often studied because it is a strict superset of first-order logic and more powerful than first-order logic, since there are things that can be proven to not be expressible in first order logic but are expressible in second order logic.

For our running example we shall express the TRANSITIVITY query, which calculates the transitive closure of a graph and cannot be expressed in FO logic, but is easily expressed in SO logic as:

$$\varphi_T(a, b) = \exists S\big(S(a, b) \wedge \forall x, y\,((E(x, y) \vee \exists z(E(x, z) \wedge S(z, y))) \rightarrow S(x, y))\big) \tag{2.2}$$

In this second order formula we first create a new relation that will store the transitive closure, then extract all the values from it ($S(a, b)$), after which we build up the transitive closure, by saying that if it is an edge ($E(x, y)$) or if there is some edge into the transitive closure $\exists z(E(x, z) \wedge S(z, y))$ then it must be part of the transitive closure ($\rightarrow S(x, y)$).

Because finding solutions to formulas written in second-order logic can be such a complex problem, often subsets of second-order logic are considered. A first example of such a subset is *existential second-order logic*, often shortened to *ESO*.

**Definition 2.1.13.** Given a second-order formula $\varphi$, it is an *existential second-order* formula if it can be rewritten to a formula of the form

$$\varphi = \exists S_1, \ldots, S_n(\psi(x_1, \ldots, x_m, S_1, \ldots, S_n))$$

where $\varphi$ is a first-order formula over a vocabulary $\sigma \cup \{S_1, \ldots, S_n\}$ with free variables $x_1, \ldots, x_m$, each $S_i$ $1 \leq i \leq n$ is a second order variable and $\psi$ becomes a first order formula with free variables $x_1, \ldots, x_m$ if we consider all those $S_i$ to be actual relations rather than relation variables.

Even though it seems like in the previous definition we pass the relations $S_1, \ldots, S_n$ as variables into a first-order formula, which is impossible, what we actually do it have $\psi$ act as if $S_1, \ldots, S_n$ are simply relations on the structures, even though in reality they are not. Examples of ESO formulas are Equations 2.1 and 2.2.

There is also a similar logic using only the universal quantifier for second-order variables, rather than the existential quantifier. This is called *universal second-order logic*, often shortened to *USO*.

Besides restricting the quantifiers of formulas in second-order logic, it is also possible to restrict the second-order variables themselves. An example of this is *monadic second-order logic*, often shortened to *MSO*, where quantified second-order variables are only allowed to be unary, which means they can only represent unary relations. The fact that these second-order variables can only be unary also means that they simply range over a subset of the universe, which makes the problem of finding these relations less complex. This restriction can also be applied to existential and universal second-order logic, creating *monadic existential second-order logic* or *MESO* and *monadic universal second-order logic* or *MUSO* respectively. Note that MSO logic is still more powerful than first-order logic. For example, in first-order logic we cannot express that a graph is disconnected, however this can be expressed in MSO logic.

### 2.1.5 Non-definability

Now that we have discussed the different logics, we shall give a short overview of some common tools used to prove whether certain queries are definable in certain logics. Also, remember that the expressive power of a logic also depends on what class of structures is being considered, not only the logic being considered. This is the same as saying that the expressive power of a logic is *context-dependent*. This makes sense if we consider $\mathcal{L}$ in $\mathcal{L}(\mathcal{C})$ a function that produces a set containing all $L$-definable formulas, as it clearly takes some class

of structures $\mathcal{C}$ as an input, implying that the output of $\mathcal{L}$ depends on the input $\mathcal{C}$. As an example, consider first-order logic, which has a very high expressive power on the structure $\boldsymbol{N} = (\mathbb{N}, +, \times)$, since every recursively enumerable relation is first-order definable on $\boldsymbol{N}$. However, when considering first-order logic on the class of all graphs, both finite and infinite, it has a rather limited expressive power, considering properties such as connectivity, which ensures that there is a path from every node to every other node in a graph, and acyclicity, which ensures that there is at most one path from each node to every other node, are both not definable in first-order logic. To prove any of these claims, model theory has three very common tools:

- the *Compactness Theorem*
- the method of *ultraproducts*
- the method of *Ehrenfeucht-Fraïssé games*

The first tool that will be discussed is the Compactness Theorem, that states that there is a model for a given set of first-order sentences if and only if for every finite subset of those sentences there is a model. One of the ways in which this is commonly used, is by first assuming there is a first-order sentence which describes something. For example, assume we have a sentence that expresses the connectivity property. Next a finite subset is created that can somehow grow easily and has a model and can easily be described in first order logic. For example, take a set of sentence that describe there is no path between two given points that is of length $n$ and that the graph is connected. The key is to choose a set of sentences that can grow easily and contains the sentence that is assumed to exist. However, when taking this set to infinity, it will contradict the sentence assumed to be true. In our example, we have a set of models where two given nodes are more than $n$ nodes apart, but the graph is still connected. Clearly each of these finite subsets has a model. However, when taking $n$ to $\infty$ it is clear that there is no model where there is no path of length 1 to $\infty$ between to given points, but the graph is still connected. Thus, there is a contradiction because the Compactness Theorem claims there must be such a model, which means that the assumption of the existence of a sentence in first order logic that describes connectivity is wrong.

Note how this only works on the set of all graphs and how nothing has been claimed about whether connectivity is definable on the set of all finite graphs. Also note that the Compactness Theorem says that every finite subset of a set of sentences needs to have a model, but puts on restrictions on this model. This means that it is very possible that each of these finite subsets has an infinite model. This thus means again that a query that is not definable in first-order logic on the class of all structures is not necessarily not definable on the class of all finite structures. The tool of ultraproducts has a similar issue with not translating easily to finite structures. For this reason there is the tool

of Ehrenfeucht-Fraïssé games, which is more easily applied on finite models and thus a very common tool in finite model theory when discussing the expressive power of first-order logic. On top of this it can be extended to study logics that are stronger than first-order logic, like the *pebble games*, which is a tool used to study the expressive power of second-order logic.

## 2.2 Ehrenfeucht-Fraïssé games

In this section, we will discuss the uses of the Ehrenfeucht-Fraïssé games in first-order logic and prove its correctness. As stated in Section 2.1.5, Ehrenfeucht-Fraïssé games is a very useful tool to express that something cannot be expressed in first-order logic. However, to be able to use this tool, we need to first define it.

**Definition 2.2.1.** Let $r$ be a positive integer, $\sigma$ a vocabulary and $\boldsymbol{A}$ and $\boldsymbol{B}$ two $\sigma$-structures, with universes $A$ and $B$ respectively. The $r$-move *Ehrenfeucht-Fraïssé game on $\boldsymbol{A}$ and $\boldsymbol{B}$* is played between two players, the *Duplicator* and the *Spoiler*, according to the following rules:
Each run of the game has $r$ moves. In each move the Spoiler plays first and picks an element of $A$ or $B$. The Duplicator in the same move then responds with an element of the universe of the other structure. An element that has been picked in a previous move may be picked again. Assume that on the $i$-th move $a_i \in A$ and $b_i \in B$ are picked with $1 \leq i \leq r$. After $r$ moves the winner of the run is decided as follows:

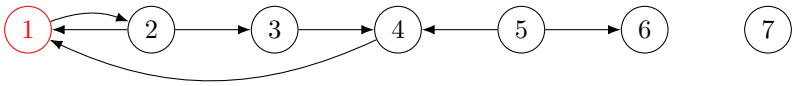
- If the mapping $a_i \to b_i$ with $1 \leq i \leq r$ and $c^{\boldsymbol{A}_j} \to c^{\boldsymbol{B}_j}$ for $1 \leq j \leq s$ is a partial isomorphism from $\boldsymbol{A}$ to $\boldsymbol{B}$, then the *Duplicator wins*
- If the mapping $a_i \to b_i$ with $1 \leq i \leq r$ and $c^{\boldsymbol{A}_j} \to c^{\boldsymbol{B}_j}$ for $1 \leq j \leq s$ is a not partial isomorphism from $\boldsymbol{A}$ to $\boldsymbol{B}$, then the *Spoiler wins*

The winner of the game is decided as follows

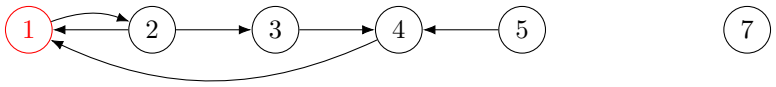
- If the Duplicator can win every run of the game, i.e. it has a *winning strategy*, then the *Duplicator wins the $r$-move Ehrenfeucht-Fraïssé game on $\boldsymbol{A}$ and $\boldsymbol{B}$*.
- If the Spoiler can win every run of the game, i.e. it has a *winning strategy* and the Duplicator does not have a winning strategy, then the *Spoiler wins the $r$-move Ehrenfeucht-Fraïssé game on $\boldsymbol{A}$ and $\boldsymbol{B}$*.

If the Duplicator wins the $r$-move Ehrenfeucht-Fraïssé game on $\boldsymbol{A}$ and $\boldsymbol{B}$, this is written as $\boldsymbol{A} \sim_r \boldsymbol{B}$.

Note that Ehrenfeucht-Fraïssé games is often shortened to EF-games, similar to FO logic with first-order logic. Also note that either the Duplicator or the Spoiler has a winning strategy, but not both or neither. Why both cannot have a winning strategy is obvious, but if the Duplicator does not have a winning




(a) The original graph of $\boldsymbol{A}$




(b) The graph to compare it to of $\boldsymbol{B}$

Figure 2.3: Two graphs that will be compared in an example EF-game. $s$ is colored in red in both these graphs.

strategy, i.e. for some run of the game it cannot win, then the Spoiler has a winning strategy by simply playing the run for which the Duplicator cannot win, since the Spoiler always gets to pick first in every move.

For our example we shall use the structures $\boldsymbol{A}$ and $\boldsymbol{B}$ for which the are graphs shown in Figure 2.3. First we will play a 1-move EF-game where the Duplicator has a winning strategy and then play a 2-move EF-game where the Spoiler has a winning strategy. For the one move EF-game, the Duplicator can clearly win: should the Spoiler chose any of the nodes 1 through 5 or 7 in either of the structures, the Duplicator simply picks the corresponding element in the other structures. Should the Spoiler choose 6 from the universe of $\boldsymbol{A}$, then the Duplicator can choose either 3, 5 or 7, since 6 in $A$ has no edges connected to 1. This is because, even though 1 has not been chosen in the game, it is always implicitly include in the partial isomorphism because it is the value represented by the constant $s$ in both structures. Thus, the 1-move EF-game between $\boldsymbol{A}$ and $\boldsymbol{B}$ can be seen as the 2-move EF game between $\boldsymbol{A}$ and $\boldsymbol{B}$ where the first move is to pick the value for the constant $s$. More generally if there are $s$ constants in a vocabulary, the $r$-move EF-game over two structures in that vocabulary is the same as the $(s+r)$-move EF game between the same structures where the first $s$ moves were picking the value of a constant and then picking the corresponding value of that same constant in the other structure.

If we wish to play a 2-move game on the structures $\boldsymbol{A}$ and $\boldsymbol{B}$, the Spoiler has a winning strategy: it starts by picking 5 from the universe of $\boldsymbol{A}$ to which the Duplicator can respond with either 3,5 or 7 from the universe of $\boldsymbol{B}$, since those are the only ones that have no edge directly to or from 1. If any other element from the universe of $\boldsymbol{B}$ is chosen, it no longer matters what nodes the Spoiler and the Duplicator choose in the next round, since when 5 in $\boldsymbol{A}$ is mapped to 1, 2 or 4 the mapping is clearly no longer a partial isomorphism. Should the Duplicator choose 3, the Spoiler will in the next move choose 2 in $\boldsymbol{A}$, which will clearly no longer result in a partial isomorphism. If the Duplicator chooses 5, then the Spoiler will choose 6 in $\boldsymbol{A}$ on the next move, which the Duplicator

clearly cannot replicate. Finally, if the Duplicator chooses 7 then the Spoiler can choose either 4 or 6 in the next move, which the Duplicator cannot replicate, since there are no edges leaving 7 in $\boldsymbol{B}$.

Note how increasing the amount of moves that can be made in our example EF-games caused the Spoiler to win the EF-game between the same structures, rather than the Duplicator. This is clearly true when $r$ becomes larger than the amount of elements in the smallest of the universes of the two structures. This is because if they are different, the EF-game will end up picking all the elements of the universe of one of the structures. Then, if the sizes of the universes are different, the spoiler picks an element out of the larger universe to which the Duplicator has to respond with an element already picked in some previous round, causing the resulting mapping to not be bijective and thus not a partial isomorphism. If the sizes of the universes are the same, the Spoiler will pick already chosen elements from one of the universe and the Duplicator will respond with the other element that was previously chosen together with the chosen element. This means that either the structures are isomorphic, in which case the EF-game will always be won by the Duplicator, since it always plays according to the isomorphism. Alternatively they are not isomorphic, in which case the Spoiler wins. This shows that how many moves it takes until the Spoiler has a winning strategy, if it is possible at all, depends on the size of the universe and, more importantly, amount of elements from the universe that actually appear in the relations. This is a fact that is often used when using EF-games to prove things, by making the structures so large that the duplicator always wins even though the two structures are different in some fundamental way.

**Proposition 2.2.1.** *$\sim_r$ is an equivalence relation on the class $S$ of all $\sigma$-structures.*

Proposition 2.2.1 follows immediately from Definition 2.2.1. This is because it is reflexive, symmetrical and transitive: the Duplicator clearly has a winning strategy if the game is played between the same structures, proving reflexivity. The Duplicator also clearly has a winning strategy for the EF-game on $\boldsymbol{B}$ and $\boldsymbol{A}$ if it has a winning strategy on the EF-game $\boldsymbol{A}$ and $\boldsymbol{B}$, since this is the exact same winning strategy. Finally, if the Duplicator has a winning strategy on the EF-game between $\boldsymbol{A}$ and $\boldsymbol{B}$ and a winning strategy on the EF-game between $\boldsymbol{B}$ and $\boldsymbol{C}$ it also has one on the EF-game between $\boldsymbol{A}$ and $\boldsymbol{C}$: if the spoiler chooses an element of the universe of $\boldsymbol{C}$, simulate the Duplicator choosing an element of the universe in $\boldsymbol{B}$, then simulate as if the Spoiler chose that same element in the universe $\boldsymbol{B}$ and actually choose the corresponding element of the universe of $\boldsymbol{A}$. When the spoiler chooses an element of the universe of $\boldsymbol{A}$ do the same swapping $\boldsymbol{C}$ and $\boldsymbol{A}$. Note that we do need to keep track of the element chose in our simulated $\boldsymbol{B}$.

In the previous examples, we saw that if at some point the Spoiler can force the Duplicator to choose an element from the universe so that the partial iso-

morphism in the end is broken because of that, then the Spoiler has a winning strategy. What causes this is that a partial isomorphism gets built up one move at a time, by for each move adding the currently chosen pair to the previous partial isomorphism. If at some point a pair is added that makes the partial isomorphism no longer a partial isomorphism, it is not possible to recover from this. This is because fixing it would require adding a different mapping for one of the two elements in the breaking pair, making the mapping no longer bijective and thus no longer a partial isomorphism. If we wish to formalize this concept we can do that with the following definition:

**Definition 2.2.2.** Assume $r$ a positive integer, a *winning strategy for the Duplicator* in the $r$-move Ehrenfeucht-Fraïssé game on $\boldsymbol{A}$ and $\boldsymbol{B}$ is a sequence of non-empty sets of partial isomorphisms $I_0, I_1, \ldots, I_r$ from $\boldsymbol{A}$ to $\boldsymbol{B}$ such that it has the following properties:

- the *forth property*: for every $i < r$, every $f \in I_i$ and every $a$ from the universe of $\boldsymbol{A}$, there is a $g \in I_{i+1}$ such that $a$ is an element of the domain of $g$ and $f \subseteq g$
- the *back property*: for every $i < r$, every $f \in I_i$ and every $a$ from the universe of $\boldsymbol{B}$, there is a $g \in I_{i+1}$ such that $a$ is an element of the range of $g$ and $f \subseteq g$

Note that in the previous definition we use $f \subseteq g$ between two functions $f$ and $g$. Hereby we mean that for every element $a$ in the domain of $f$, it is also in the domain of $g$ and that $g(a) = f(a)$. Also note how the fourth property defines that the Duplicator must have a good move when the Spoiler picks some element from the universe of $\boldsymbol{A}$ and how the back property says the same if the Spoiler picks some element from the universe of $\boldsymbol{B}$.

### 2.2.1 Quantifier rank

Now that Ehrenfeucht-Fraïssé games have been introduced and claimed to be related to first order logic, it is time to start to relate them more directly. Before we can do this however, we need to introduce an important concept of a FO formula: its quantifier rank.

**Definition 2.2.3.** Assume that $\varphi$ is a first-order formula over some vocabulary $\sigma$, the *quantifier rank of* $\varphi$, denoted by $qr(\varphi)$, is the depth of the quantifier nesting in $\varphi$, inductively defined on the construction of $\varphi$ as follows:

- If $\varphi$ is *atomic*, which means it is either a relation symbol from $\sigma$ with constants or variables filled in or a variable and some constant or two variables compared to each other with $=$, then $qr(\varphi) = 0$
- If $\varphi$ is of the form $\neg\psi$ then $qr(\varphi) = qr(\psi)$
- If $\varphi$ is of the form $\psi_1 \wedge \psi_2$ or $\psi_1 \vee \psi_2$ then $qr(\varphi) = max\{qr(\psi_1), qr(\psi_2)\}$

- If $\varphi$ is of the form $\exists x(\psi)$ or $\forall x(\psi)$ then $qr(\varphi) = qr(\psi) + 1$

As an example of quantifier rank $\exists x_1, x_2((\forall x_3(E(x_1, x_3))) \wedge E(x_1, x_2))$ has a quantifier rank of 3, since it increases one per quantified variable. The formula $\exists x_1(\exists x_2(E(x_1, x_2))) \wedge (\forall x_3(\neg E(x_3, x_1))))$ has a quantifier rank of 2, even though there are 3 quantified variables, but because of the $\wedge$, the max is taken of those two subformulas which is 1. Also note that if there is a formula $\varphi$ with $qr(\varphi) = r$ then for every $r' > r$ there is an FO formula $\psi$ such that $qr(\psi) = r'$ and that $\varphi$ and $\psi$ are logically equivalent. This can clearly be done by adding an existential quantifier over some new variable.

### 2.2.2 Equivalence

Now that the quantifier rank has been defined we can use this to define an important equivalence relation. This relation is important because it will allow us to define the relation between the first-order logic and the Ehrenfeucht-Fraïssé games.

**Definition 2.2.4.** Assume that $r$ is a positive integer and $\boldsymbol{A}$ and $\boldsymbol{B}$ are two $\sigma$-structures. Then $\boldsymbol{A}$ and $\boldsymbol{B}$ are *$\equiv_r$-equivalent in first-order logic* if they satisfy the same set of first-order sentences of quantifier rank $r$. This is written as $\boldsymbol{A} \equiv_r \boldsymbol{B}$.

**Proposition 2.2.2.** *$\equiv_r$ is an equivalence relation on the class $S$ of all $\sigma$-structures.*

Proposition 2.2.2 follows directly from the definition of $\equiv_r$-equivalence. To start off, $\equiv_r$-equivalence is symmetrical, since if $\boldsymbol{A}$ and $\boldsymbol{B}$ satisfy the same set of FO sentences of quantifier rank $r$, then so do $\boldsymbol{B}$ and $\boldsymbol{A}$. Next is the reflexivity, which clearly holds, since $\boldsymbol{A}$ and $\boldsymbol{A}$ satisfy the same set of FO sentences of quantifier rank $r$. Lastly, transitivity also holds. Assume that $\boldsymbol{A}$ and $\boldsymbol{B}$ satisfy the same set of FO sentences of quantifier rank $r$ and that we call this set $S_{AB}$. Also assume there the same for $\boldsymbol{B}$ and $\boldsymbol{C}$ and call this set $S_{BC}$. Now since $\boldsymbol{B}$ can only have one set of all FO sentences of quantifier rank $r$ it satisfies, then $S_{AB} = S_{BC}$. This clearly shows that $\equiv_r$-equivalence is an equivalence relation.

Notice how when defining $\equiv_r$-equivalence, we didn't simply define equivalence under all FO sentences but under those of quantifier rank $r$. The reason for doing this, is that it allows us to relate it to an $r$-move EF-game. This relation is defined in the following theorem, which allows us to use an EF-game to prove something cannot be expressed in first-order logic.

**Theorem 2.2.1.** *Assume that $r$ is a positive integer and $\boldsymbol{A}$ and $\boldsymbol{B}$ are two $\sigma$-structures. The following statements are equivalent:*

- *$\boldsymbol{A} \equiv_r \boldsymbol{B}$, which means $\boldsymbol{A}$ and $\boldsymbol{B}$ satisfy the same set of FO sentences of quantifier rank $r$*

- $\boldsymbol{A} \sim_r \boldsymbol{B}$, *which means that the Duplicator has a wining strategy for the* $r$*-move EF-game on* $\boldsymbol{A}$ *and* $\boldsymbol{B}$

Theorem 2.2.1 clearly shows that we can use an $r$-move EF-game to prove that a sentence of quantifier rank $r$ will give the same result on both $\boldsymbol{A}$ and $\boldsymbol{B}$. This is because if the same sentences are true on both structures, clearly the same sentence must say either true for $\boldsymbol{A}$ and $\boldsymbol{B}$ or say false for $\boldsymbol{A}$ and $\boldsymbol{B}$.

### 2.2.3 Usage

Now that we have defined the equivalence between $\equiv_r$-equivalence and the spoiler winning the $r$-move EF-game, we can use this to prove that certain queries cannot be expressed in FO logic. To do this we will for the given query define two structures $\boldsymbol{A}$ and $\boldsymbol{B}$. We choose these structures so that the query is true in one of them but false in the other. We then show that for an $r$-move EF-game we can create these two structures $\boldsymbol{A}$ and $\boldsymbol{B}$ so that the Spoiler has a winning strategy. If we do this for any $r$ then we have proven that the formula for this query cannot exist in first order logic. The reasoning is that we have show that the formula cannot be uniformly defined. This is because no matter how many quantifiers we allow it, there will always be a pair of structures where it must return the same result, even though it should be different.

The $\equiv_r$-equivalence only considers first-order sentences, and not formulas that have free variables. A possible way to solve this issue is to create a Boolean problem that could be solved using the free variables of the formula. We would then proceed to prove that our new Boolean problem cannot be expressed in the first-order logic, as described in the previous paragraph. After this, we would prove by contradiction that our original query cannot be written in first-order logic. To do this, we describe our new Boolean problem and describe a first order logic formula, using the original query. This means that if we could write our original query in FO logic, we could do the same for our new Boolean problem. However, we know that our new Boolean problem cannot be written in FO logic. Thus, our original query must not be able to be written in FO logic, because if it did, it would create a contradiction.

As an example of this last technique, assume that we wish to prove that the query that returns all pairs of nodes in disconnected components of a graph cannot be expressed. We will call this query the different components query. What we would do is we would use the Boolean problem of whether a graph is a single connected component. We assume that a formula $\varphi_{connected}$ exists that is true only if the structure it is evaluated in is a graph with a single connected component. Next we prove that this formula cannot exist using an EF-game and the following contradiction. We can now prove that the different components cannot be expressed. We first assume that it can be expressed by the formula $\varphi_{diff-comp}(v_1, v_2)$ which is true if $v_1$ and $v_2$ are nodes in different component of the graph. Now we show that the connectedness query can be solved using the

different components query by defining $\varphi_{connected} \equiv \exists v_1, v_2(\varphi_{diff-comp}(v_1, v_2))$. This formula clearly would solve the connectedness query, but we have already proven that it cannot be expressed in FO-logic. Thus, our assumption that the different components query could be expressed in FO-logic must be wrong.

# Chapter 3

# Code Pattern Mining

In code pattern mining we attempt to extract some often recurring patter out of some large code base. Typically, this is not done by trying to find patterns in the text, but it is done on an abstract syntax tree, which stores code in a tree structure so that it is clearer what is related to each other and in which way. This gives us a better context and thus makes it easier to extract useful patterns that on their own make sense in a programming language. However, the structure from which we attempt to extract our patters is only one part of the puzzle: we also need to define how we find what are common patterns and how we attempt to choose the most useful of these patterns. Thus, in this chapter we shall start with defining what an abstract syntax tree is. Then we shall define some concepts that will be used when defining the similarities. Next we shall use these concepts and AST's to define different types of similarities within it. After that, we shall use these different kinds of similarities to define different types of algorithms that can be used to extract these similar patterns from an abstract syntax tree.

## 3.1 Trees

### 3.1.1 Abstract Syntax Tree

When trying to find patterns in code we typically begin by creating an *abstract syntax tree* of the code, often shortened to *AST*. This is an abstract representation of the source code where the children are part of the parent. As an example, say we have the while loop of Listing 3.1. This fragment of code could then be converted to a tree like the one given in Figure 3.1. Notice how this example defines everything without requiring the order of the children. This is because for each node, no two children have the same label. If the order of the children is available, we can compress our tree down to the tree in Figure 3.2. In this tree we have that the condition of the while loop is always its left child and the body is the right child of the while loop. We also have the infix expression,

which simply written in the order in which it appears, rather than explicitly having children named 'LeftOperand', 'Operator' and 'RightOperand'.

```
while (a > 1) {
        a = a + 1;
}
```

Listing 3.1: A simple example of a piece of code to be parsed

The reason to use an AST rather than just text is mainly that its structure is useful. For example, a code miner working purely on strings might pick up on the fact that 'while (a > 1){' is a commonly occurring substring, but it will have trouble understanding that a while-loop always needs a condition and a body. For the AST we can also define a context-free grammar, which is a set of rules that define what the children of a certain node with a certain label can be. This is also very convenient since it allows us to describe in a very compact way what makes a piece of code valid. This can be very useful when attempting to do code search by generating an AST. On top of this, when looking for patterns in an AST, we will always get something that can easily be converted into syntactically correct code. This however only works if we assume we take a node and its entire maximal subtree underneath it and convert that AST. This generated code may not be valid on its own in a file but if given to the compiler it could be parsed correctly. As an example, assume we found a common pattern of a getter function in Java. It would need to be placed in a class to be made valid, but it makes sense on its own and could be successfully parsed, just not compiled. On the other hand, if we were to look for common patterns in the code as a string we might find patterns that, while they commonly reoccur, do not make much sense on their own. As an example, assume we find a pattern like 'while (a > 1){' to be very common. The compiler could however not parse this into a valid AST, since it is an incomplete piece of code: we are missing at least a '}'. This thus shows how using an AST to describe a pattern will result in more useful code. This is because it could be converted to code, and it is guaranteed that the compiler will be able to parse this code. However, this is not the case when we look for common string patterns, as we have shown.

### 3.1.2 Labeled Ordered Trees

Now that we have explained what kind of structure we will be pattern mining over, it is time to define the structure formally. This structure will be a *labeled ordered tree* on a set of labels. This means that we take a set of labels, for example all the labels that can appear in an AST, and make a tree using those. We simply define that each node must have a label, but do not prescribe any meaning or enforce any kind of structure. More formally we define it as follows:

**Definition 3.1.1.** Assume $\mathcal{L}$ is a finite set of labels $l_0, \ldots, l_n$. A *labeled ordered tree on* $\mathcal{L}$ is a 6-tuple $T = (V, E, \mathcal{L}, L, v_0, \preceq)$. Here $(V, E, v_0)$ defines a tree with

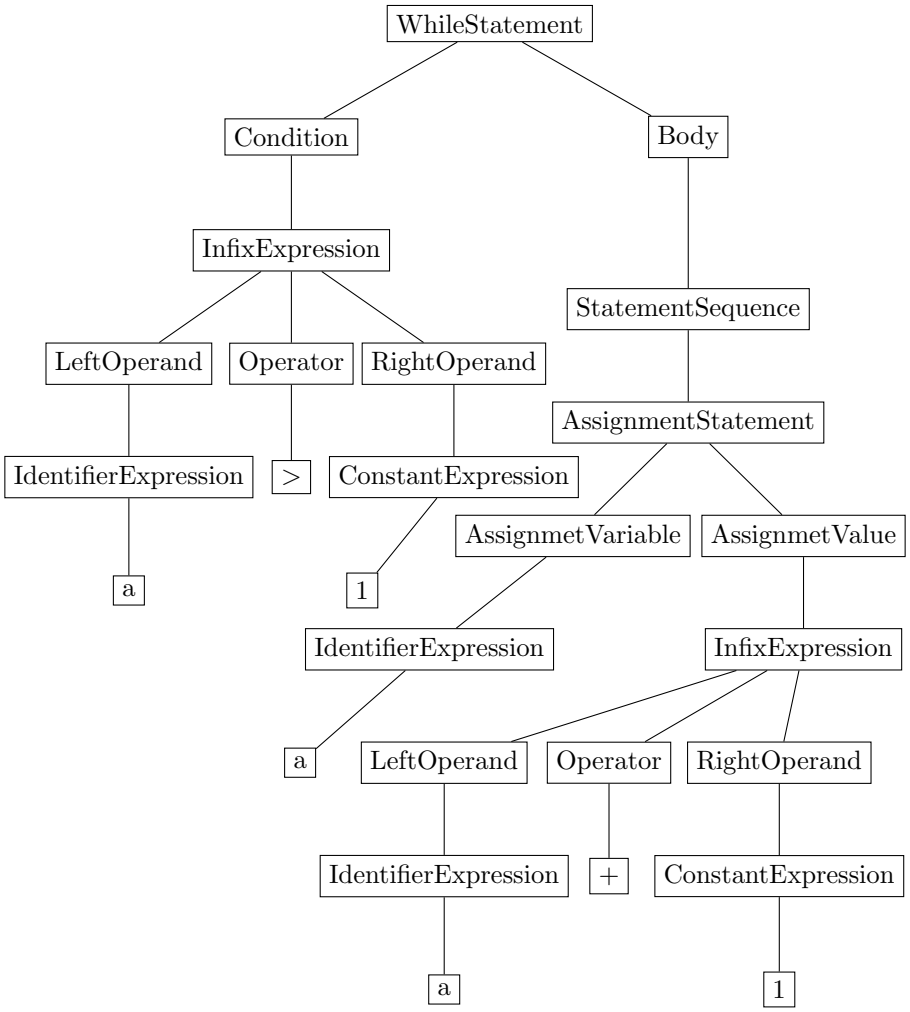



Figure 3.1: The AST of the code in Listing 3.1

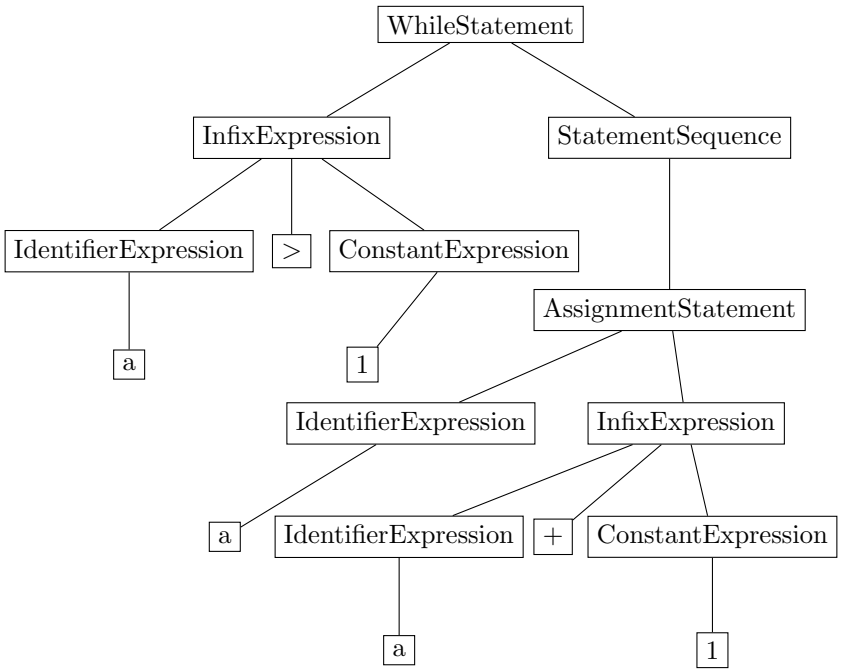



Figure 3.2: A more compact AST of the code in Listing 3.1

$V$ the set of nodes and $(u, v)$ $inE$ the set of pairs where $u$ is the parent of $v$ and $v$ is the child of $u$. $\mathcal{L}$ is the set of labels that can be applied to a node. $L : V \to \mathcal{L}$ is the labeling function, giving a label from $l \in \mathcal{L}$ to each node $v \in V$. $\preceq$ is the elder sibling relation, that gives all pairs $(u, v)$ where $u$ and $v$ have the same parent but $u$ is left of $v$ or the same node. The size of a tree $T$ is $|T| = |V|$. A labeled ordered tree is said to be in *normal form* if $V = \{1, \ldots, k\}$ and the nodes are numbered by the preorder traversal of the tree. If there are multiple trees, we write $T = (V_T, E_T, \mathcal{L}_T, L_T, v_{0_T}, \preceq_T)$.

Notice how $(V, E, v_0)$ is enough to define a tree. With this we can identify the root of the tree, the different layers of the tree and distinguish between different nodes. Notice how the root could also be defined as the node which has no parent node. If we then want to add labels to it, we need to add both the set of labels $\mathcal{L}$ and the labeling function $L$. This then assigns a label to each node, allowing us to find different nodes or groups of nodes that have the same label. Note how $\mathcal{L}$ must contain at least the labels used in the tree, but can also contain other labels not used in the tree. If we want to make the tree ordered, we add the elder sibling relation $\preceq$ to allow us to identify the order between nodes on the same level. This then allows us to define what is the order of siblings of the same parent.

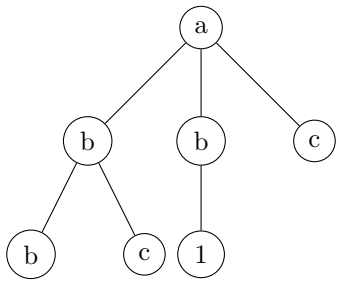



Figure 3.3: An example of a labeled tree

As an example of a labeled ordered tree, consider the tree in Figure 3.3. We will now write a complete 6-tuple that defines this tree. To start, we will number the nodes layer by layer, from left to right. This means that node with label 'a' will be 1, first child of node 1 will be node 2, the second child of node 1 will be 3 and so on. This thus gives us the set of nodes $V = \{1, 2, 3, 4, 5, 6, 7\}$. Next we will define the root node $v_0 = 1$. After this, we will give our tree structure by defining its edges. This will result in the relation $E = \{(1, 2), (1, 3), (1, 4), (2, 5), (2, 6), (3, 7)\}$. Now that we have defined the basic tree structure, we will make our tree more recognizable by adding labels to it. First we shall define the set of labels $\mathcal{L} = \{a, b, c, 1\}$. Notice how we have a 1 both as a label and as a node. It is important to notice how these describe completely different things: one represents a unique node and the other a label that can be applied to a number of nodes. Now that we have given the set of labels we shall define which node gets which label. The labeling function becomes $L : V \to \mathcal{L} = \{(1, a), (2, b), (3, b), (4, c), (5, b), (6, c), (7, 1)\}$. Note that, because $L$ if a function, a node can only have a single label. Finally, we need to define the elder sibling relation to give the order of our nodes. This would result in the relation $\preceq = \{(1, 1), (2, 2), (2, 3), (2, 4), (3, 3), (3, 4), (4, 4), (5, 5), (5, 6), (6, 6), (7, 7)\}$.

The previous tree we defined was not in normal form. To show how multiple different definitions can give the same tree, we shall define the tree in Figure 3.3 but in normal form. For the set of nodes we get $V = \{1, 2, 3, 4, 5, 6, 7\}$, with the first child of the first child of the root node being 3. The edges of the tree would be $E = \{(1, 2), (1, 5), (1, 7), (2, 3), (2, 4), (5, 6)\}$. The set of labels would be the same as the previous example. The labeling function would become $L : V \to \mathcal{L} = \{(1, a), (2, b), (3, b), (4, c), (5, b), (6, 1), (7, c)\}$. Finally, the elder sibling relation becomes redundant, since if the number of the node is lower than or equal to a different node and if they share a parent, it is the elder sibling of that node. However, for completeness we shall include it. The elder sibling relation is $\preceq = \{(1, 1), (2, 2), (2, 5), (2, 7), (3, 3), (3, 4), (4, 4), (5, 5), (5, 7), (6, 6), (7, 7)\}$.


Figure 3.4: An example of a subtree

### 3.1.3 Subtrees

Now that we have defined AST's we can start to look for patterns in them. These patterns are simply a set of nodes that occur frequently in a particular shape together in a tree. That is to say that they create a tree of their own which can be found in the larger tree. This concept is called a *subtree*. There are many definitions of a subtree, so we shall give one and use this to define similarities.

**Definition 3.1.2.** Assume $T$ and $t$ are labeled ordered trees. $t$ is a *subtree* of $T$ if $\mathcal{L}_t \subseteq \mathcal{L}_T$, every node in $V_t$ can be mapped to a node in $V_T$ and this mapping then also maps the values of $E_t, L_t$ and $\preceq_t$ to values of $E_T, L_T$ and $\preceq_T$ respectively.

If we consider $\mathcal{L}_t \cup V_t$ as the universe for $t$ and similarly for $T$, then the subtree definition simply requires that there is a partial isomorphism between $t$ and $T$. More specifically, this partial isomorphism must be between $t$ and a substructure of $T$. Additionally, it must map the values of $\mathcal{L}_t$ to the same values in the universe of the substructure of $T$.

As an example of a subtree, consider the tree in Figure 3.4. Assume we call this tree $t$. Then we would have the relations $V_t = \{1, 2\}$, $E_t = \{(1, 2)\}$, $\mathcal{L} = \{a, b\}$, $L = \{(1, a), (2, b)\}$, $\preceq = \{(1, 1), (2, 2)\}$. If we want to figure out whether this is a subtree of Figure 3.3we attempt to define the mapping. Assume we call this tree $T$ and that it is in normal form. Then the mapping from $t$ into $T$ simply maps node 1 of $t$ to node 1 of $T$ and node 2 of $t$ to node 2 of $T$. Alternatively, we could also map node 2 of $t$ to node 5 of $T$. This clearly shows that a subtree can have multiple locations it can appear in the larger tree. This will become important when mining, as it our goal is to find trees that are often a subtree of the AST.

### 3.1.4 Pattern trees

When attempting to pattern mine, there will be two types of tree: the subtrees we are attempting to find and the tree in which we are trying to find them. We call the first kind of trees *pattern trees* and the other tree the *data tree*. Typically, a code pattern mining algorithm will want to find those pattern trees that are a subtree at many locations in the data tree. If a pattern tree is a

subtree of a data tree, we say it can be *mapped* into the data tree. A mapping $\varphi : P \to D$, also known as a *matching function*, of a pattern tree $P$ onto a data tree $D$ is a function that maps every node in the pattern tree to a unique node in the data tree and thus is injective. It also has some additional requirements. Firstly it must preserve the parent relation, meaning that if a node $a$ is a parent of a different node $b$ in the pattern tree, $\varphi(a)$ must also be a parent of $\varphi(b)$ in the data tree. On top of this the elder sibling relation must be kept. This means that if a node $a$ is an elder child of a node $b$, then $\varphi(a)$ must also be an elder sibling of $\varphi(b)$. Finally, the matching function must also preserve labels, which means that if a node has a label in the pattern tree, the node it is mapped to in the data tree must have the same label. This means that for a node $a$ in the pattern tree, $L_P(a) = L_D(\varphi(a))$. A mapping function is thus the formal version of the partial isomorphism described in Section 3.1.3.

To determine how frequent a pattern tree $P$ is in a data tree $D$ we need to determine the concept of *occurrences*. We say a pattern tree $P$ *matches* a data tree $D$ or $P$ *occurs* in $D$ if there is a matching function $\varphi P \to D$. The *total occurrence* of $P$ in $D$ with regard to $\varphi$, assuming that $P$ is in normal form and has $k$ nodes, is $Total(\varphi) = (\varphi(1), \ldots, \varphi(k)) \in (V_P)^k$. This means that the total occurrence is just another way of writing $\varphi$ if the pattern tree is in normal form. The *root occurrence* of $P$ in $D$ with regard to $\varphi$ is the node $Root(\varphi) = \varphi(1) = \varphi(v_{0_P}) \in V_D$, assuming that $P$ is in normal form for $\varphi(1)$. Finally, we define the *set of root occurrences of $P$ in $D$* to be $Occ(P) = \{Root(\varphi) \mid \varphi \text{ is a matching function of } P \text{ into } D\}$, the size of which we will use to define the frequency of a pattern in a data tree $D$. Thus, the *frequency* of a pattern tree $P$ in a data tree $D$ is $freq_D(T) = |Occ(T)|/|D|$. For some positive number $0 < s \leq 1$, a pattern tree $T$ is said to be *s-frequent* in a data tree if $freq_D(T) \geq s$.

## 3.2 Types of similarity

In pattern mining the end goal is to find patterns that frequently occur. However since we do not expect patterns to reoccur verbatim, we have to define some metric indicating whether it is the same pattern or not. This is called defining similarity, since we define when two patterns can be considered similar. Since the concept of similarity is an intuitive one that could be approached from multiple angles and is rather subjective, there are also multiple definitions of similarity. It is also often the case that these different definitions are used by different algorithms. The choice of which definition to use, and often thus which algorithm to use, depends on what will be done with the mined patterns, as some definitions are focussed on a particular use-case, improving the quality and usefulness of the mined patterns for that use-case.

In the following sections we shall introduce pattern tree similarity, the most commonly used similarity among the algorithms we will describe. It uses the

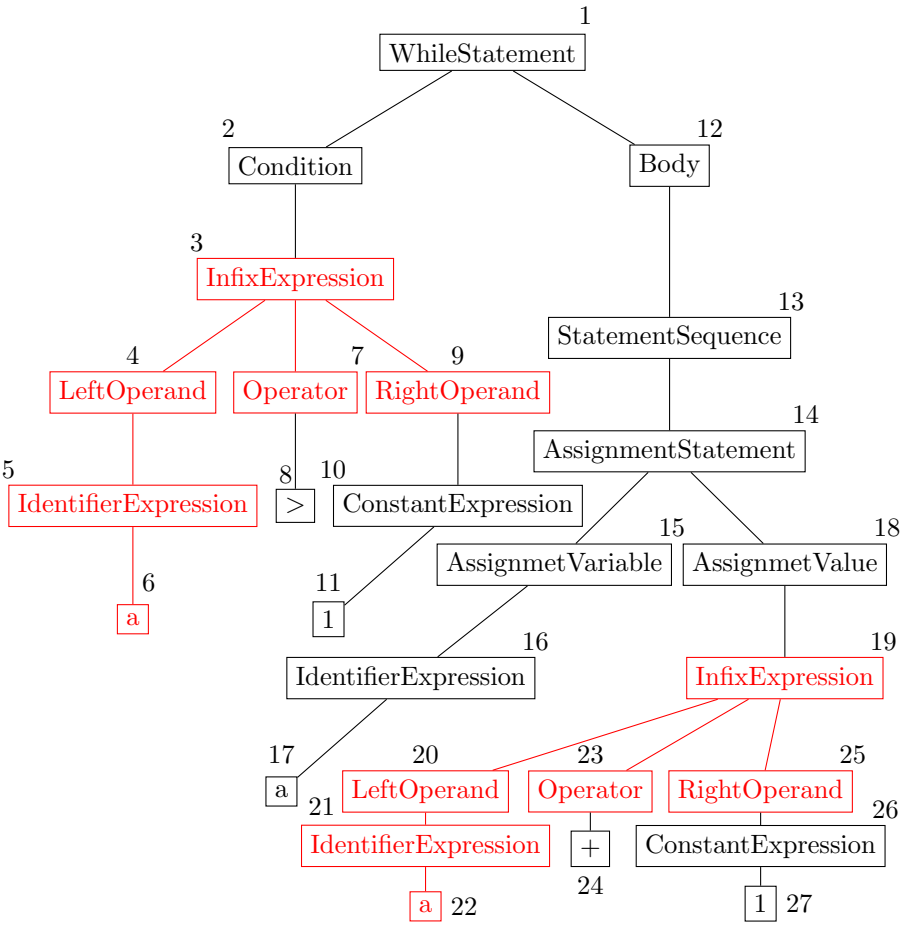
previously described pattern trees to define similarity between two subtrees of the AST. After that we will also discuss the API set and the API sequence similarities. These similarities mainly consider the function calls a particular piece of code makes, rather than the structure of subtree of the AST. Lastly we shall also consider defining similarity by using an intermediate language and translating our AST or subtrees of our AST.

### 3.2.1 Pattern tree similarity

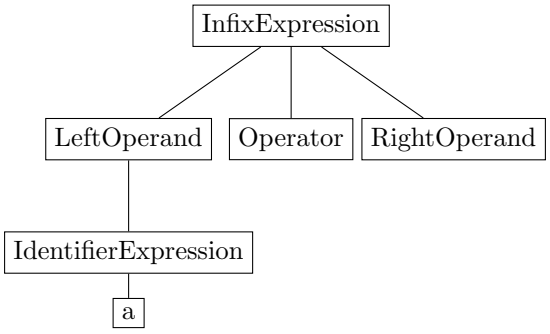
The first kind of similarity comes from the matching defined by the frequent tree mining algorithm. In this algorithm we use the concept of pattern trees to define when a particular subtree is similar to another. We say that two subtrees of the data tree are similar if they have a pattern tree that has a mapping to each of the two subtrees. We call this kind of similarity *pattern tree similarity*. Consider now only the subtree of the data tree, whose nodes are exactly the nodes of the pattern tree after being mapped. The matching function is now also a surjective function, since it maps at least one node to every node in the subtree. Since it is both injective and surjective the mapping is bijective. This is the case for both matching functions if we restrict the data trees only the subtrees to which they are being mapped. This means that there is also a bijective function between the two restrictions of the data tree. The bijection also implies that these two structures are isomorphic to each other. However, if we define the universe as the union of the nodes and the labels, this isomorphism defined by the bijection must map the labels to themselves. This is very similar to the partial isomorphism discussed in Section 3.1.3.

Assume we have only pieces of code and want to consider whether they are similar or not. We would start by converting each of these codes to their AST's. Then we would consider whether we could define a bijective function, as described above, between these two trees. If there is such a bijective function, we could then always find a pattern tree by simply copying one of the AST's and defining it the pattern tree. The matches between the two AST's and new pattern tree are the found bijective function and the identity function of the other.

As an example, consider the tree in Figure 3.5a to be the data tree. The two highlighted subtrees in it are clearly pattern tree similar, since they can clearly both be mapped into by the pattern tree in Figure 3.5b. Assume both the data tree and the pattern tree are in normal form. The matching function from the pattern tree to the left subtree is then $\varphi = \{(1,3),(2,4),(3,5),(4,6),(5,7),(6,9)\}$. The matching function of the pattern tree to the right subtree is $\varphi = \{(1,19),(2,20),(3,21),(4,22),(5,23),(6,25)\}$. From Figure 3.5a it is also very clear to see how there will be an isomorphism between the two subtrees.




(a) An example of an AST two similar subtrees highlighted, with their node numbers




(b) A pattern tree

Figure 3.5: An example of two similar subtrees and their pattern tree

### 3.2.2 Maximal subtree similarities

The next few types of similarity can only be used on code that is an entire function or a subset of it and thus is not as broadly applicable as the previous type of similarity. However, this is still an interesting approach to take and can still be extracted from AST's and thus is useful for our case. For all of these approaches we will take pieces of code that can be parsed by a compiler to an AST. If we receive only the AST, this is the same as for a given node taking the largest subtree that can be found with that node as its root. For a given node $v_0$, we call such a tree the *maximal subtree with root node* $v_0$. Thus, when talking about a *code fragment* of our AST, we will be talking about a piece of code whose AST forms a maximal subtree with some root node in our AST. We will often represent our examples as lines of code as this is compacter and more intuitively understandable.

#### API set similarity

A first of these similarities is *API set similarity*. In API set similarity we enforce that the set of function calls is the same for a given code fragment. This thus means that as long as the same sets of functions are called they are considered similar. Thus, they can be similar regardless of their order or whether they are in conditional blocks or not. If the AST of the entire program were available, it would also be possible to recursively calculate the set of function calls that go outside the program to some other libraries. However, this requires more preprocessing and cannot be derived purely from two code fragments. This and other function call-based similarities are thus most useful on programming languages that have an extensive standard library or smaller projects that all use a particular library. The reason being that it is more likely that a pattern will use the functions form these libraries, rather than functions they have written on the spot for essentially the same functionality. This will then mean that the functions called will match more often.

```
int a = countA();
if (a > 0){
    printA();
}
```

Listing 3.2: A code fragment to compare for similarity

```
int a = countA();
if (a == 0){
    printA();
}
```

Listing 3.3: A second code fragment to compare for similarity

```
int a = countA();
```

```
printA();
int b = countA();
```

Listing 3.4: A third code fragment to compare for similarity

```
int a = countA();
int b = countB();
if (a > 0 && b> 0){
    printA();
}
```

Listing 3.5: The final code fragment to compare for similarity

As an example, consider the Listings 3.2 to 3.5. For Listing 3.2 the set of API calls is {countA, printA} and for Listing 3.3 this is {countA, printA}. For Listing 3.4 the set of API calls it makes is {countA, printA} and Listing 3.5 calls the set of {countA, printA, countB}. In API set similarity, Listing 3.2, 3.3 and 3.4 are all similar. This can be clearly seen when looking at the set of API calls made by the fragments. Notice how the API set similarity does not care about conditional blocks or whether certain API calls are repeated or not. It only considers that the API call can happen.

### API sequence similarity

The next type of similarity we will consider is the *API sequence similarity*. The API sequence similarity is similar to the API set similarity, because both only consider API calls. However, in this case we say that two AST's are similar if their possible sequences of API calls match. This is thus a stricter version of the previous similarity type. The considerations made for the previous type, regarding its usefulness and ways of extending it beyond what is strictly contained in the AST, can also be made here.

As an example, take the listings of Listings 3.2 to 3.5. For Listing 3.2 the set of all possible API sequences is {(countA, printA), (countA)} and for Listing 3.3 this set is {(countA), (countA, printA)}. The set of possible sequences of API calls for Listing 3.4 is {(countA, printA, countA)} and for Listing 3.5 it is {(countA, countB, printA), (countA, countB)}. In this case only Listing 3.2 and 3.3 are considered similar. Notice how they were considered similar, even though their conditional block does the opposite, but their set of possible sequences of API calls is the same. Also notice how repetitions are also considered in this similarity, for example in Listing 3.4.

### Intermediate language similarity

A final type of similarity that will be discussed here is *intermediate language similarity*. In intermediate language similarity we consider two AST's equivalent

if their projection into an AST of some intermediate language is similar under pattern tree similarity. This has the benefit that certain aspects that if there are parts we do not consider important that they can be projected away by not being present in the intermediate language. Depending on what parts of the language are projected away and in which manner, this can also make finding similar patterns significantly less computationally complex since the input can become simpler. This is also clearly the most flexible kind of similarity since it depends on the intermediate language used and how the projection into this language happens. It also has the added benefit of making it possible to compare AST's of different languages. If we compare AST's that use different labels, we can simply define translations for both of them, and now it is possible to consider similarities between the different languages. However, it should also be noted that this same approach would also be possible if we apply the translation as a pre-processing step, where we do the translation before checking any similarities. This can have the side effect of increasing the amount of similarities we find. This change in frequency depends on how the translations are done to the intermediate language.

As an example, take the AST of Figure 3.5a. Let us define the translation so that every 'Operator' node and its child are translated into simply the 'Operator' node. Let this translation also translate any 'IdentifierExpression' and its child to simply 'IdentifierExpression' and the same for 'ConstantExpression'. This then translate our tree to the one in Figure 3.6. Notice how the patterns that were similar under pattern tree matching in the old tree, are still similar in the newly translated tree. Also notice how this translation essentially claims that it does not care what variables are used in operations, simply that they reference variables.

## 3.3 Approaches

As shown by the different types of similarity there are multiple ways to approach and interpret mining code patterns. This also leads to different algorithms using different kinds of similarity and different additional restrictions in the hopes of having it return less results but having those results be more useful. There are two main groups of approaches that will be discussed here: those that mine directly on the AST of the code, the so-called *tree miners*, and those that mine using an intermediate language.

### 3.3.1 Tree miners

A first category of miners for code patterns are the tree miners. These miners typically work on any kind of tree, but some have been optimized to give the best results when mining on trees that are AST's. Firstly we will discuss the more general *FREQT* miner algorithm that mines patterns from trees in general. After that we will discuss the *FREQTALS* miner algorithm which is an extension

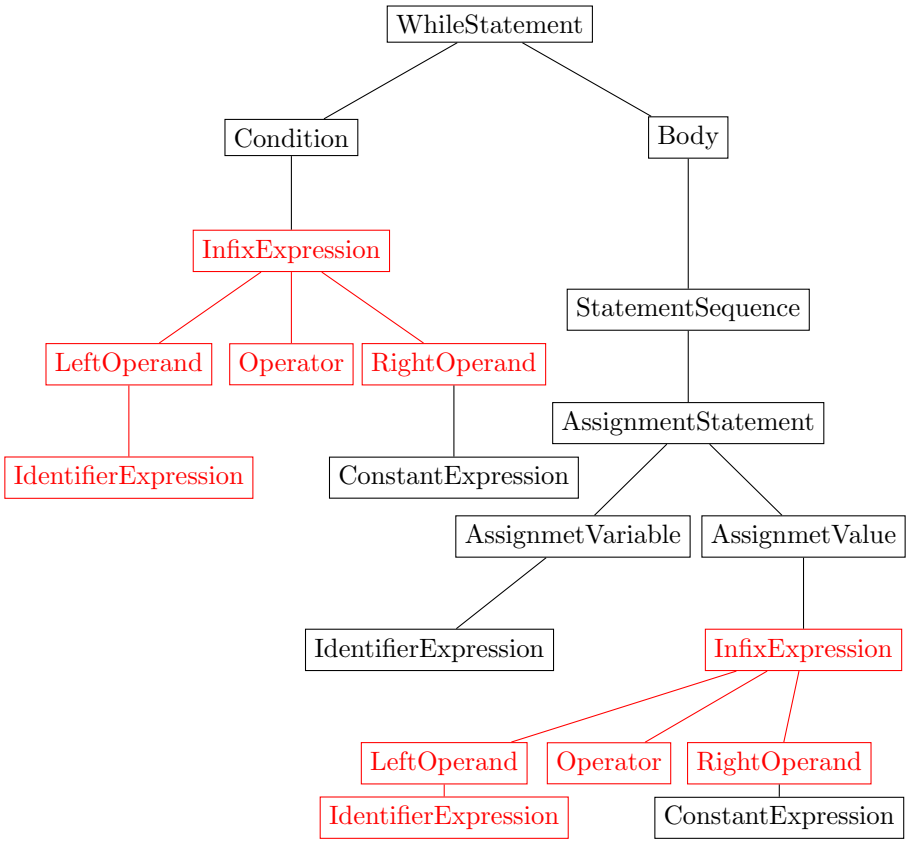



Figure 3.6: An example of an AST translated into an intermediate language

of the FREQT algorithm that is optimized to give fewer but more useful results for end user and is designed specifically around detecting patterns in AST's of code.

### FREQT

The first algorithm we shall define is the FREQT algorithm. It attempts to find all pattern trees that are s-frequent for a given s in a given tree. The reason for discussing this algorithm is that it is both an influential and a broad algorithm. This allows us to first define a rather broadly applicable but relatively efficient algorithm and then later on specialize it for our application. However, even without this specialization, FREQT returns all frequent pattern trees and thus solves the question of code pattern mining. Thus, most of the section on FREQT will be based on the original paper by T. Asai et al. [AAK$^+$02].

The approach that FREQT takes to tree pattern mining is similar to the approach that is often used in association rule mining, where first all the frequent single element sets are checked and then those are recombined to two element sets, which are then recombined to three element sets etc. FREQT starts off by creating a set $\mathcal{F}_1$ of all pattern trees of size 1 that are s-frequent. It does this by traversing the data tree $D$ and storing all their occurrences in $RMO_1$, thus having a list of occurrences per pattern. In subsequent passes of FREQT, if it is the $k$-th pass with $k \geq 2$, FREQT will incrementally compute a set $\mathcal{C}_k$ of all *candidate patterns* of size $k$. After that it will compute the set $RMO_k$ of rightmost occurrences for all candidate patterns. A *rightmost occurrence* of a pattern tree $P$ in a data tree $D$ with regard to a matching function $\varphi$ is the node $Rmo(\varphi) = \varphi(k)$ of $D$ where the rightmost leaf $k$ of $P$ maps to. If $P$ is in normal form, then $k = |V_P|$. The set of rightmost occurrences can thus be defined as $RMO(P) = \{Rmo(\varphi) \mid \varphi \text{ is a matching function of } P \text{ into } D\}$.

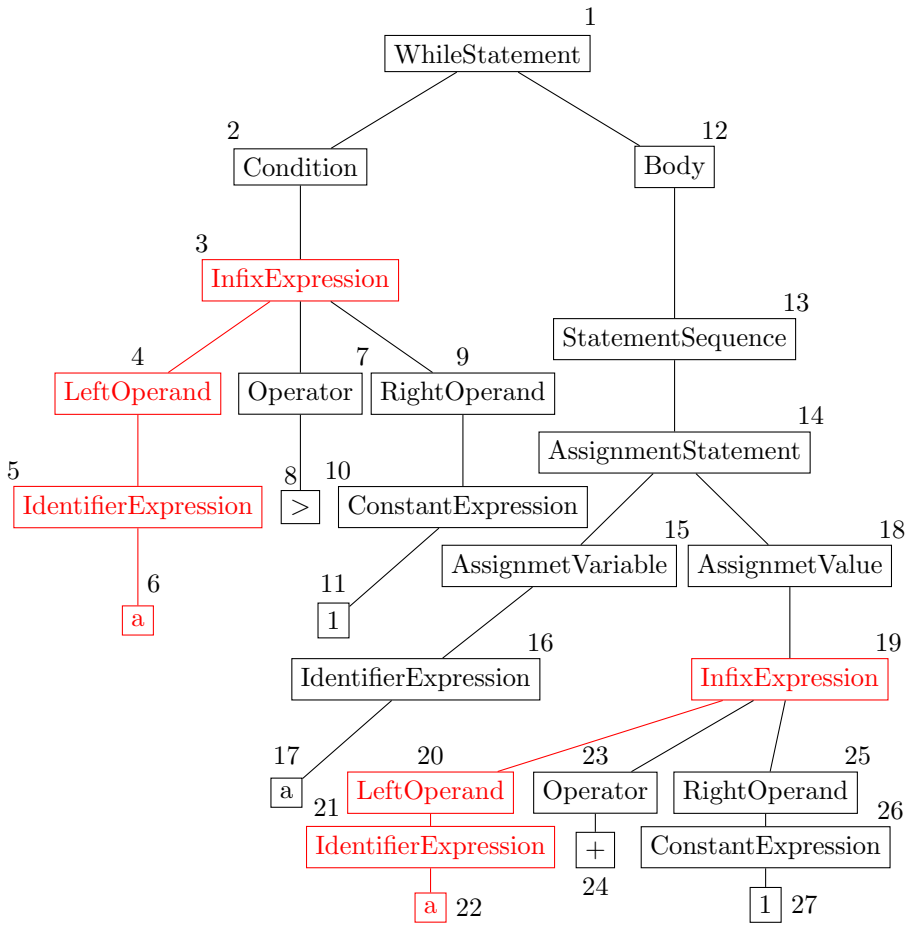
The set of rightmost occurrences is similar to the set of root occurrences in the sense that it has a clear limit on how many of them there can be. However, due to FREQT making extensive use of the rightmost expansion, it is easy to deduce the set of rightmost occurrences from the previous one. On top of this it is easy to calculate how many root occurrences a pattern tree has from its rightmost occurrences. If we look at the pattern tree, we know what the length is of the path from the root node to its rightmost child. If we then take all the rightmost occurrences and go up the same amount of nodes for each rightmost occurrence we get the set of root occurrences. Alternatively, we could also keep track of the root occurrences, but this would be redundant since we have to keep track of the rightmost occurrences anyways. The reason is that the rightmost occurrences are used to check for the amount of patterns possible for the expansion of the pattern tree. If we did not have these, we would have to recalculate them from the root occurrences every step, leading to a lot of double work.

FREQT works by looking at the pattern trees and then adding a rightmost expansion. A *rightmost expansion* of a tree $P$ is the addition of a node with some label as the rightmost child of a node somewhere along the rightmost branch. The rightmost branch is the path between the root node and the rightmost child, which is the last child when traversing the tree in preorder. Using a rightmost expansion has the benefit that there is always exactly one tree of which the current tree is the rightmost expansion. This is in contrast to when adding labels in arbitrary locations, where one tree could be created from multiple different trees. On top of this, with rightmost expansion all trees that could be created by adding nodes in arbitrary places can still be created. This thus means that we can consider all trees to be organized in a tree where if the tree $S$ is a rightmost expansion of a tree $T$, that $T$ is the parent of $S$. FREQT could then be described as simply doing a breadth-first search through this tree. For a given rightmost expansion we can also efficiently calculate all their rightmost occurrences from their previous ones. We do this checking for each rightmost occurrence, all the right siblings of the node which would be a left sibling of the new node. We then check if any of these nodes in the data tree have the correct label. If such nodes exist with this correct label, we can save those nodes in our list of rightmost occurrences of the expanded pattern tree.

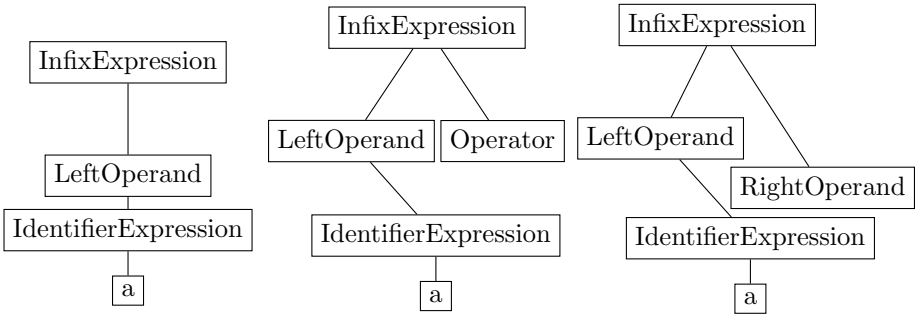
The algorithm also does duplicate detection to make sure it does not check the same node in the same data tree multiple times. This duplicate detection works efficiently since all right most occurrences with the same parent are listed after each other in $RMO_{k-1}$ thanks to the way the update of $RMO$ scans all children of the same parent to see if they could generate a new rightmost occurrence.

As an example, consider the AST in Figure 3.7a. In this tree we have highlighted both occurrences of the pattern tree from Figure 3.7b. Assume that the pattern tree is also in normal form. The rightmost occurrences of this pattern are $\{6, 22\}$. Assume that we are adding a node with label 'Operator' to our pattern tree. We would then attempt to add it as a child underneath node 4 of the pattern tree and add it as a new rightmost child underneath nodes 1, 2 and 3. However, the only expansion that actually occurs in this AST is the one where the node becomes the new rightmost child of node 1 of the pattern tree. This expansion is shown in Figure 3.7c. We can then check for the new rightmost occurrences of it by going two nodes up from the previous rightmost occurrences. In this case that means we go up to the nodes 4 and 20 in the data tree. Then we check those nodes' right siblings. We then find nodes 7 and 23 which match the label. These are then added to the rightmost occurrences of the expanded tree. We can do the same procedure for the expansion shown in Figure 3.7d. This results in the new set of rightmost occurrences of 9 and 25.

Now that we have defined how FREQT works, there are some small optimizations that can be made to generate fewer candidates without missing any of the frequent ones. These techniques are somewhat reminiscent of the techniques




(a) An example of an AST two similar subtrees highlighted, with their node numbers




(b) A pattern tree

(c) A rightmost expansion of the pattern tree

(d) Another rightmost expansion of the pattern tree

Figure 3.7: An example of two similar subtrees and their pattern tree and two expansions of that pattern tree

used to speed up associate rule mining. When doing a rightmost expansion of a tree, we only need to consider those labels that have a node with that label in $\mathcal{F}_1$, thus skipping some labels every time we expand a pattern tree. This is similar to how association rule mining can be sped up by looking at the previous set of frequent associations and only generating those sets of associations of a size one larger for which each subset of one size smaller was also frequent. Similarly, FREQT can also be sped up by keeping $\mathcal{F}_2$ and only generating expansions which result in the new rightmost child $r$ to have a parent node $p$ such that the tree with $p$ the parent of $r$ also appears in $\mathcal{F}_2$. This means that in any candidate there are no pairs of parent and child nodes $p$ and $t$ that is not also a tree with $p$ the parent and $t$ the child node in $\mathcal{F}_2$. If there were such a pair the tree could not be frequent since if the subtree is not frequent, the entire tree will not be frequent. A similar logic could be applied with more complex tree with $n \geq 3$ nodes, but this tends to become so computationally complex that it's benefits are severely diminished.

Finally, it should be noted that FREQT returns every s-frequent tree. That means that if a pattern tree is s-frequent it and all its subtrees are returned by FREQT. Whilst this is entirely correct, the smaller patterns are less useful to a human who has to interpret these pattern trees representing code patterns. As a result, the FREQT algorithm was expanded upon to create the FREQTALS algorithm.

### 3.3.2 FREQTALS

Now that FREQT has been discussed, it is clear that it is good at efficiently calculating all pattern trees that are frequent in some data tree. However, it is not good at selecting what patterns will be useful to a human that is attempting to understand these patterns. If we decide to thus add additional restrictions to what trees we would like to see returned, it would also be likely that we could make the algorithm itself more efficient by taking these restrictions into account to prune more pattern trees that would never be returned. This is the approach taken by FREQTALS that we described earlier: we specialize the FREQT to work with code pattern mining. We thus specialize FREQT by adding additional constraints to it. This is exactly what K. Mens et al. [PNM+19] did in the paper defining FREQTALS. As a result, most of the section explaining FREQTALS is based on their paper.

FREQTALS adds the idea of constraint-based data mining to the FREQT mining algorithm by enforcing 8 constraints numbered $C0$ through $C7$. The first constraint is $C0$ which is a constraint already enforced by FREQT and simply made explicit in FREQTALS, namely the *minimum support constraint* which enforces that all patterns must have a minimum support of some fixed value. This is the same as enforcing that all trees must be s-frequent for a particular $0 < s \leq 1$. The value of this $s$ can be determined from the size of the data tree

and the required support. Next is the *maximum size constraint* $C1$ where we limit the amount of leaf nodes our tree can have. This constraint goes together well with the *minimum size constraint* $C2$ that requires that every tree needs to have at least a certain amount of leafs. Note how both of these are simply, so the mined pattern is useful but not so large a human has difficulty understanding it or so large it would take the computer too long to compute it. Also note that while $C1$ can be used to limit the amount of pattern trees that are being searched, $C2$ can only be used to limit the trees being output, but not the ones being searched.

Next are the constraints on the labels. Firstly, there is $C3$ which limits the set of labels that are allowed to occur in the root of the mined patterns. This allows the mining algorithm to limit the set of pattern trees it needs to explore based on their root labels. However, if we still wish to use the optimizations we defined for FREQT by using $\mathcal{F}_1$ and $\mathcal{F}_2$, then we will need to ignore this rule when mining for patterns of size one and two. We will need to still filter out the patterns that do not adhere to $C3$ when outputting all found patterns. This constraint also makes a lot of sense for a human. It makes sense to return a pattern that is rooted at a 'ForLoopStatement', but not one that is rooted in a 'Condition', should we have a more explicit AST like in Figure 3.1. The next constraint on labels is $C4$ which restricts pattern trees to not contain a specific set of labels. This again makes the search tree for the algorithm smaller since it no longer needs to check any pattern tree in which any node uses any of these labels. Note that this restriction can be applied when calculating $\mathcal{F}_1$ and $\mathcal{F}_2$ to be used with the optimizations, since any tree with those labels could not be used to expand any pattern tree anyways. From a user perspective this clearly also makes sense if there are certain kinds of patterns we do not wish to find. For example, we may not be interested in patterns about the class definition. Next there is constraint $C5$, which limits the number of siblings that can have the same label. This again can limit the amount of pattern trees that need to be searched. From an end user this also makes sense as for example a high amount of repetitions of the same node would likely express a long list of something being defined, like for example a static array with pre-defined values.

As a final label constraint, there is also a restriction on the labels of leaf nodes: $C6$. This restriction dictates that all leafs in the pattern tree must have labels that also occur as leafs in the data tree. This also allows pruning of the search space. We can prune by realizing that at any point all the leafs of the pattern tree except for the rightmost child will always be leafs. Thus, if a rightmost expansion makes a node a leaf that is not allowed to be a leaf, this expansion does not need to be checked. This restriction also makes sense from an end-users perspective, since if they would want to understand the code pattern mined, they could attempt to convert it to code again. This can only be guaranteed to be possible if the pattern has leaf nodes with labels that are also the labels of leaf nodes in the actual data tree.

The last constraint added by FREQTALS is $C7$, which enforces that all nodes with a specific label must also have a certain set of obligatory children with specific labels. This requirement however is only enforced on nodes with *structural labels*. A label is considered structural if in the data tree for in each occurrence there are never two children with the same label and if for all pairs of occurrences the labels that their children share are in the same order. The set of *obligatory children* for a node with a specific label are then the set of child labels that all the occurrences of the label in the data tree have in common. Similar to $C6$ we can use this constraint to prune certain pattern trees from our search space as soon as we know that a sibling can no longer be added to a node, and they do not have all required children. This definition also means that this restriction requires the use of the more elaborate way of writing an AST like in Figure 3.1, rather than the more compact notation of Figure 3.2. However, for an end user again this makes a lot of sense since this for example would be able to enforce that for a while loop, both the body and the condition need to be contained in the pattern tree.

Finally, the last addition made by FREQTALS to FREQT is the *maximal subtree mining* where it takes the results produced by FREQT with the added constraints of $C0$ through $C7$ and then expands these trees as much as possible while keeping the same set of occurrences. This essentially allows FREQTALS to search through the set of possible pattern trees fairly quickly due to $C1$ using FREQT and then expand these best results as much as it can before it becomes less frequent. This in combination with $C1$ allows for a trade-off between the computational efficiency and the size and complexity of the pattern trees.

This thus shows that FREQTALS is an extension of FREQT that attempts to provide additional constrains that are useful in the context of an AST but may not be so useful for tree mining in general.

### 3.3.3 Intermediate languages

The previous approaches both rely on finding frequent patterns directly in the AST. Alternatively, we could use some intermediate language to translate our AST to and then pattern mine on that AST. This has the added benefit of being relatively flexible, since the language can be chosen to accommodate only the parts that are of interest, but this also means that for each language a full translation must be devised. It also has the benefit that if the intermediate language is simpler than the original that, if we simply executed a normal tree mining algorithm on it, it would just take less time to execute than on the normal AST.

This approach is not as common in the standard code pattern mining setting, however it can be found in adjacent problems, such as contextual code search. Contextual code search is the problem of attempting to suggest a piece of code,

typically a function, based on the surrounding context, for example the documentation around it or the name of the function or its parameter types. A relatively efficient solution to this problem was proposed by Mukherjee et al. in the form of their CODEC system [MCJ20]. In this system they focus on searching for entire functions from context clues of the function and its surroundings, which they accomplish by training a neural network to generate an AST of an intermediate language from a given context. It then also stores all AST's of functions in the intermediate language with their context and the original code in a database. Finally, when a context is given and a function for that context is requested, CODEC takes that context to generate an AST in the intermediate language and then looks up similar pairs of AST's in the intermediate language and context in the database and then returns the associated program code.

It is thus clear that the CODEC system also attempts to find pairs of similar code patterns. However, unlike in frequent tree mining, it does not restrict the searched patterns to only the frequent ones, but also considers the less frequent ones. On top of this rather than only focus on the tree patterns that have to be similar, it also considers the surrounding context in an attempt to better guide the search of similar pairs of code and because in their application the code to which we are attempting to find the most similar pattern to is not known. Since this problem is only adjacent to our discussion here, we shall extract the interesting concepts from it, namely using an intermediate language, and apply those to code pattern mining.

With regard to the intermediate language there are two ways in which this can be used in the mining algorithm: either as a pre-processing step followed by a normal frequent tree mining algorithm, which will potentially run more efficiently if the intermediate language is simpler than the original language. The other way in which this can be used is by integrating it directly into the algorithm and adding the frequency of all patterns that have the same AST in the intermediate language together as if they are the same pattern.

### Preprocessing AST's

The first way to use the intermediate language is to translate to it before mining patterns. Thus, we start with to defining the intermediate language. Then define a translation from an AST in the current language to the intermediate. Next, as a pre-processing step, we translate the entire data tree to an AST of this intermediate language. Finally, we run FREQT or FREQTALS or any other tree miner on this AST and these give the frequent patterns of the AST in this intermediate language. However, this means that the found frequent patterns are AST's in the intermediate language and not in the in original language. Depending on the intermediate language this might be good enough. For example if an AST in the intermediate language translates easily to some pseudocode that can be understood by a programmer, then the algorithm could simply output this pseudocode and that might be sufficient.

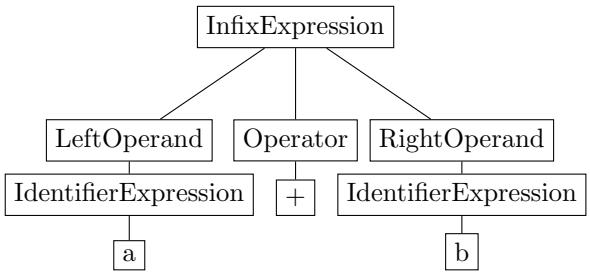
As an example, consider the translated AST of Figure 3.6. Assume that 'ConstantExpression' was also part of our found pattern. In this case we could attempt to translate our subtree into pseudocode, but this would result in a very vague translation. We would get something along the lines of 'var op const'. If this were the original AST, we could have returned something along the lines of 'a op 1'.

It is also possible that pseudocode is not good enough for the application or that, as we showed in our example, that the intermediate language does not translate well into pseudocode. If this is the case, a possible solution would be to generate a possible or set of possible AST's of the original language from the pattern. Again these could also be translated into an actual piece of code if this is what is needed. The reason for giving a set of possible patterns is because it is likely that multiple slightly different AST's in the original language get represented by the same tree in the intermediate language. This means that when we are given a single pattern in the intermediate language that there are likely many possible trees that it could be in the original language. Thus, to give a better idea of what the pattern was that was found, it might make sense to return multiple patterns in the original language so the end-user can see what the commonalities are between these or choose which is most appropriate. However, generating these AST's is not a trivial task, since it is most useful if these patterns are translated back to pattern trees that actually occur in the data tree. This could be done relatively efficiently if we know the occurrences of the tree in the data tree in the intermediate language and if we can map the nodes from the intermediate language AST back to the original AST. This would allow us to generate total occurrences in the intermediate language AST which we could then map back to the original data tree.

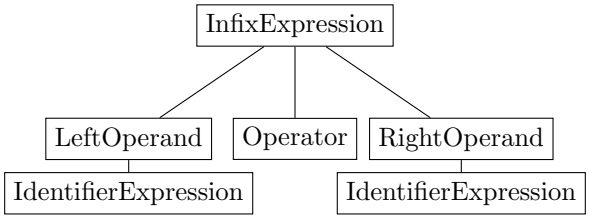
As an example, consider our pattern from the previous example again. If we actually kept track of its origins, we could find out that it came from two AST's which represent the expression 'a $<$ 1' and 'a + 1'. Instead of thus generating all possible combinations like 'c = 5' and 'b − 15', we could then simply return these patterns. This would then also indicate that this pattern always seems to be applied to the variable 'a' and always has the constant '1' in its right-hand side. If we simply returned a possible set of information we would understand that it is a variable, some operator and a constant, but we would not notice this extra information.

#### Translating during mining

Instead of trying to find the occurrences of a pattern in the intermediate languages afterwards, we can simply use the intermediate language to relate multiple patterns as being the same. This also prevents us on potentially missing out on some frequent patterns because multiple occurrences in the original data tree get converted into a single one in the data tree of the intermediate lan-




(a) A simple AST




(b) A translated AST

Figure 3.8: An example of an AST and its translated counterpart

guage. However, this comes at the cost of adding complexity and thus losing the efficiency gain of the previous approach.

As for how this would be implemented, it would mostly function similarly to FREQT or FREQTALS but with the size of the pattern being determined by the size of its tree in the intermediate language. This gives us the question of how to generate a new pattern that is one size larger. The straightforward solution of generating new patterns in the intermediate language and then translating these into the original language, will not work. This is because it is very possible that the translation will provide an infinite amount of patterns. Even if the translation does not provide an infinite amount of patterns, the amount of patterns it will generate will be very large, which would be rather inefficient. A better alternative would be to generate patterns on the original tree, similarly to how this happens normally in FREQT. We would then translate these into an AST in the intermediate language to compare which patterns are considered similar. For all the patterns considered similar this way, they would all contribute their root occurrences to a count of the root occurrences shared between these patterns, but keep their own rightmost occurrences to allow for easy expansion in the same way as FREQT normally does. However, if they have no occurrences they are immediately removed from the set of candidates to check for the current size.

As an example consider the AST in Figure 3.8a. Assume that within this AST we were considering the pattern trees consisting of an 'IdentifierExpres-

sion' and its child 'a' and another one but with its child 'b'. Under normal pattern tree similarity these are not considered similar. However, when we look at the translated AST in Figure 3.8b, we can see they are both mapped to a single 'IdentifierExpression' node. Thus, in the translated AST they are considered similar under pattern tree similarity. This means that when the algorithm will consider whether these patterns are frequent, both will be considered to have 2 root occurrences. This is because all similar patterns essentially pool their occurrences when the algorithm tries to determine their frequency. However, each of these pattern trees only has one rightmost occurrence, namely the occurrence of their 'a' and 'b' nodes. These will then be used when attempting to expand these patterns.

Even through we expand the trees similar to FREQT, how the expansion works regarding moving to trees of a larger size is still different. This is because what we consider the size of the pattern trees to be, is the size of the pattern tree translated to the intermediate language. The algorithm would attempt to expand the pattern tree in the original language with a rightmost expansion, just like in FREQT. If this expansion results in a tree in the intermediate language that is one node larger than the size of all current pattern trees in the intermediate language, it gets added to the list of candidate pattern trees. However, it is possible this results in a translation to the intermediate language where a set of nodes could not be translated. This would likely be because a translation requires may require multiple nodes to generate a single node in the intermediate language. If it has some node that could not be translated, then we add it to the list of candidates to check for the current size of pattern trees. A third possibility is that the candidate results in a translation with an extra node, but also with some untranslated subtree. If this is the case, we would remove it from the candidates of the current size.

At the end of all expansions, if there is nothing new in the list of candidates to check for the current size, then it moves on and checks the candidates of ones size larger. If there are still new candidates in the list of candidates to check for the current size, the algorithm checks that these all have at least one occurrence. This is easy to do since it can simply check the occurrences from the preceding pattern tree that this pattern tree is an expansion of. This checking is very similar to how we do it with normal rightmost expansion. Note how these incomplete translations do not contribute their occurrences to any pattern, since they do not translate properly to some AST in the intermediate language. However, we do need to still keep track of them to make occurrence checks of expansions still efficient. After this check, the algorithm attempts to expand these new trees again as previously described. Also note how we remove candidates where they result in an extra node in the translation but also still have some untranslated subtree. This is because this indicates that a subtree was being explored to generate a node, but then the rightmost node moved further up in the tree, which left this subtree only partially explored. Because

the rightmost child has moved to a location that no longer allows exploration of this subtree, we ignore this pattern tree in favor of the one where this subtree was never explored to begin with.

As an example, consider again the AST of Figure 3.8a. Assume that here we have found the pattern of size two containing 'InfixExpression' and 'LeftOperand'. On possible expansion of this is by adding the 'IdentifierExpression' underneath the 'LeftOperand' node. This would result in a tree that does not translate to our intermediate language. This is because we only translate an 'IdentifierExpression' and its child node to and 'IdentifierExpression' node. Thus, when checking if this pattern is frequent enough, this pattern will get expanded upon again. If we expand this pattern with 'RightOperand', it will get removed the next time we check frequencies. This is because while adding 'RightOperand' causes the size to go to 3, it still leaves 'IdentifierExpression' untranslated. Because the rightmost occurrence is now on the 'RightOperand' branch, the 'IdentifierExpression' could never be expanded to become fully translated. Alternatively, if we expand the pattern we currently have with 'a' this will result in an AST that has no untranslated nodes and has 3 nodes in its translation. This is then added to the candidate set of the next size.

The previous paragraph describes how FREQT could be adjusted to use an intermediate language, but there are optimizations that can be made. Firstly the tricks used in FREQT to make it easier to skip certain expansions can be used here as well, but it then checks that the expansion results in a translation so that the intermediate language pattern trees that can generate a frequent tree according to the optimizations. If the expansion results in a new node in the translation that isn't in a translation of the patterns in $\mathcal{F}_1$ then it is skipped. Also, if the expansion results in a new edge being created in the translated tree that does not appear as an edge in the translations of the patterns in $\mathcal{F}_2$, it is also skipped. A new optimization that could be made is one caused by the expansion when trying to find a new tree that will translate into a pattern tree with one additional node. In this process it is possible that one subtree gets partially explored and then the expansion happens in a position that this subtree can no longer be explored. This can be prevented by keeping the last tree that was used as a basis for this tree that translated to tree without any untranslated subtree. If the expansion on the pattern tree ever happens so that, when the base tree is removed there are two disconnected subtrees, this expansion can be ignored. Finally, the largest problem that needs optimization is that the algorithm often keeps expanding a subtree that does not contribute to adding a new node in the translation since it is removed entirely. To solve this we could when defining the translation also define a function that is given the entire subtree that is being expanded and cannot be translated currently and that returns whether the subtree rooted at the rightmost child will contribute to any translation or not. This allows our translation to be arbitrary and to use the knowledge of the human defining the translation to still increase our

efficiency.

This second approach could also be used to implement the API set similarity by simply only keeping the function calls from the AST and some other structures such a function declarations such that it is clear where the boundaries of a function are. The API sequence similarity could be implemented similarly, where only the function calls are kept in the translation, together with certain nodes that indicate a change in control flow, such as if blocks or for loops.

# Chapter 4

# Relational Meta-Algebra

In the previous chapter we discussed multiple algorithms to mine patterns from an AST. Mining a pattern is essentially querying the code for similar patterns to a given pattern. There are many algorithms and languages to ask queries in general, but the most common one is SQL. SQL is a well studied language and finds its foundations in the relational algebra. Of course a query in relational algebra could also be considered a piece of code and could be turned into an AST. This AST could then also be mined. However, there are also algebras designed to allow us to query relational algebra queries themselves. In this chapter we shall start by looking at an extension of SQL that allows us to query trees that represent AST's of SQL queries. Next we shall discuss the foundation of SQL: the relational algebra. This allows us to form a more mathematical representation of a query. After that, we introduce its logical counterpart: the relational calculus. With the relational calculus we will be able to prove what queries and cannot be expressed in it and the relational algebra. Once both the relational algebra and the relational calculus have been introduced, we consider how to expand both, so they can query and execute relational algebra queries. This then results in the relational meta-algebra and the relational meta-calculus.

## 4.1 Meta-SQL

The first type of query language we shall discuss is *Meta-SQL*. This is an extension of the SQL languages that allows us to look into queries stored in the database as if they were a data structure, defined by Van den Bussche et al. [dBVV05]. It thus allows us to inspect queries. It does this by storing all queries in an XML format. However, to inspect this XML data we require some other query language, as SQL can only query the data in its own table. For this we use the XSLT language. XSLT is a language designed to transform XML document and is a W3C standard [Kay17]. Finally, they also add an evaluation function, which takes an AST of an SQL query and executes the query it represents. The reason we choose to consider this language is that it is a

very practical language, considering it is an extension of SQL. On top of this it will allow us to introduce a recurring concept and demonstrate its usefulness. Finally, it allows us to highlight the strengths and weaknesses with an approach where existing technologies are combined versus extending a piece of existing technology with a new technology.

### 4.1.1 Adding XSLT to SQL

The first thing that is changed in Meta-SQL compared to standard SQL that we will discuss, is the addition of the XML datatype. After this, we will discuss the power of adding the querying of this XML datatype to SQL with Meta-SQL. This allows Meta-SQL to inspect and transform the documents that are of an XML datatype. Next we will discuss how Meta-SQL adds to SQL the part that ties the XML querying and transformation together with SQL: XML variables. Finally, we will also discuss one final addition that makes it easier to work with XML variables: XML aggregation. This allows a set of XML values in an XML variable to be combined into a single value. In the following paragraphs we shall discuss all these additions, the reasoning behind their addition and the pros and cons of them.

**XML Datatype** The first addition to SQL is the introduction of the *XML datatype*. This on its own is not anything special. This is because without anything that interacts with this datatype, it would not make a difference if it were stored as a string in the database. However, once other tools are introduced that can interact with the datatype in a meaningful way, having an XML datatype becomes essential. If we did not have such a datatype, we would be forced to use a string column and then check that a given string is a correctly formatted XML document before interacting with it.

**XSLT** The next addition of Meta-SQL to SQL is the use of *XSLT*. In the previous paragraph we discussed an XML datatype but had no way to interact with it. With the introduction of XSLT we can actually manipulate the data within the columns with an XML datatype. XSLT is a very powerful XML transformation language, which takes as an input an XML document to operate on and optionally some parameters. These parameters can be supplied from the SQL query, thus allowing us to interact with the XML document by manipulating it based on the values in some tables.

XSLT operates by recursively applying templates starting at the root of the input document. Each template has an associated XPath expression that specifies on which nodes of the input template it should be applied. XPath is another language specifically designed for XML and allows us to select certain sets of nodes. The result of an XPath expression is always a set of nodes in the XML document tree. The XSLT template to be applied at any particular time will be the first one in the document to return a non-empty set of nodes from its XPath

expression. When applying a template, it is also possible to give an XPath expression. If one is given, a template will be applied to each node that the XPath query returned. On top of this it is also possible to give each template a mode and when applying a template to only consider templates of a given mode. There is also other functionality, such as a 'for each' loop, conditional statements and tree variables. All of this complexity also shows the potential problem with using XSLT in a query language: XSLT is Turing-complete [Kep04]. This means that it is entirely possible to define queries for which we shall never get the answer. It also means that it is difficult to optimize a query containing an XSLT query, as we cannot know the size of its output beforehand. However, it also grants us a great flexibility in how we wish to inspect and transform the stored XML documents. To make use of an XSLT transformation, we define a function in the database that contains an XSLT document. This consists of XSLT templates and an optional a set of parameters to be referenced within any of the templates. Note that an XSLT document does not need to return an XML document. For example, it can also return a document that contains only the text node of a string. This can then be converted by the database to a string rather than an XML type.

**XML variable** Now that XSLT has been introduced, we can transform documents. However, we cannot do anything yet with these transformed documents. For this Meta-SQL introduces *XML variables* to SQL. Where normal SQL variables range over the rows of tables, XML variables range over subelements of an XML document. An XML variable can be introduced in a FROM-clause similar to a how the rows of a subquery is introduced. For an XML variable $s$ definition would be $xiny[e]$, where $y$ is either an XSLT function call or a previously bound XML variable and $e$ is an XPath expression. $x$ is then filled in with each node the XPath expression returned. This then behaves similar to if $y[e]$ was a subquery that returned a row for each node returned by the XPath expression $e$.

**XML aggregation** The final addition of Meta-SQL to SQL is *XML aggregation*. This is very useful as it provides the inverse operation of the XML variable. XML aggregation allows us to combine multiple XML documents into a single one. This is done with the natural aggregation function CMB. In CMB all the functions given to it will be inserted into a new tree as a child of a new root node labeled 'cmb'. This is essentially the XML counterpart of the SUM aggregation for number types. Of course if we want a different label than 'cmb' we can simply use the query doing this aggregation as a subquery and then apply an XSLT function to its value.

### 4.1.2 Evaluating AST's

So far we have talked about the features added by the Meta-SQL to help it process XML. However, we have yet to describe what the entire reason is we wish

to work with XML: we want to work with AST's as XML documents. Now that we have defined a powerful method to interact and transform XML documents, we can use this to also interact with AST's of other SQL queries. Assume we have some table `Views` which contains a pair of view names and the queries associated with them. One possible way to use what we have already defined, is to update all views that refer to a specific relation to use a new relation. Of course this is only a very basic query, but it shows what the potential power could be of what we have defined so far. However, the strongest and arguably the most powerful aspect that the Meta-SQL has not yet been discussed: the *evaluation function*.

The evaluation function allows us to go from *syntactical meta-querying* to *semantical meta-querying*. In syntactical meta-querying we only consider the syntax of the inspected queries. In semantical meta-querying on the other hand, we consider what the results of a query actually are. To do this we introduce the EVAL function. This function takes as its input an XML document representing the AST of the SQL query to execute. Then as its output it returns the rows of the executed query. Because of this, EVAL can only appear wherever a reference to a table would be able to appear. This concept can become even more powerful by the introduction of EVAL's untyped counterpart: UEVAL. UEVAL returns a table of XML documents rather than an actual table. Each row in this table is a root node with for each column the query would have returned a child with the name of that column. These children only have a single child, which is the value of that column in that row. UEVAL can appear in the same locations in our queries that EVAL could. However, the power of UEVAL lies in that it allows us to execute queries with different output schemas without having to have a different set of columns for each query executed. The values of UEVAL can only be extracted by an XML variable, compared to a standard variable for EVAL.

### 4.1.3 Considerations

Now that we have explained what Meta-SQL is and what it can do, we need to consider the pros and cons to it. The single most useful addition made to SQL by Meta-SQL is the introduction of EVAL and UEVAL. These functions allow us, in combination with the power of XSLT, to evaluate any arbitrary query at runtime. It would be possible to argue that the fact that XSLT is Turing-complete negates this need, but XSLT does not have an easy way to access the tables of the database. So while XSLT could be used to program EVAL, it would not be able to return any values from tables of which the values are not known beforehand.

One of the biggest issues with Meta-SQL is also one of its biggest strengths: it uses XSLT. At the time Meta-SQL was proposed there were already many optimizers and executors of XSLT. This means that it would be relatively easy

to integrate into an existing database system. On top of that it also means that a lot of the work for how to optimize these queries has already been done and does not need to be researched. However, because XSLT is Turing-complete it is also very difficult to optimize and it cannot be guaranteed that it will give an answer. This is very problematic as it means that a single query could lock up an entire database until some timer kills it or a user manually stops the query. Of course this is always a concern for any large query but in this case it is possible the query never ends. A normal large query on the other hand is at least guaranteed to find a solution at some point.

A final consideration to make is that almost all the XML additions were also done by SQL/XML. SQL/XML uses XQuery rather than XSLT however. In the paper defining Meta-SQL, the authors also noted that should XQuery be more desirable than XSLT this would be relatively easy to swap out. However, the original paper for Meta-SQL was submitted two years before the introduction of XQuery to SQL/XML. On top of this, while there is a significant overlap between the two, switching to XQuery instead of XSLT does not alleviate the problem of it being Turing-complete [Kep04].

## 4.2 Relational algebra

Now that we have discussed a system that uses SQL as a basis and then extends it to allow for manipulation of stored queries, we shall look at the foundation of SQL. In following sections we will then expand upon this foundation to again allow for manipulation and execution of stored queries. However, to be able to understand and expand upon this foundation we first need to understand it. This foundation is called the *Relational Algebra* and is often shortened to *RA*. The relational algebra is a language that allows us to define queries over a database procedurally. That is to say that, when defining a query in the relational algebra, we state how we transform the input relations to get an output relation. This is in contrast to the relational calculus, which we will discuss in Section 4.3, which is declarative. This means that is simply describes what the result should be like and not how to get it. However, such a declarative query must, and in this case can, always be translated to a procedural language to be actually executed and get the result. We shall go more into this in Section 4.3.

The relational algebra thus defines a set of operators that allow us to define queries over a database. However, before we can define these operators and how they behave, we must define what they operate on. In the following section we will define the structures over which the relational algebra is evaluated. We will also compare it to the definitions given in Section 2.1.1 to make it easier to understand.

### 4.2.1 Databases

To be able to define the database over which a relational algebra expression will be evaluated, we first need to define the structure of the database. This is done by defining a *database schema* which contains all the relation names and the arities of these relations. On top of this it could also contain the names of the columns of the relations. However, in our case we shall simply access these different columns by referring to the column number.

**Definition 4.2.1.** A *database schema* $D = \langle R_1, \ldots, R_n \rangle$ is a fixed set or $n$ relation symbols $R_1, \ldots, R_n$. Each relation symbol also has some *arity* associated with it. Assume that for every relation symbol $R_i$ its arity is $a_i$. This would be written as $R_i : a_i$. The arity defines the amount of columns each row in an instance of a relation symbol will have.

Notice how the database schema is similar to the vocabulary defined in Definition 2.1.1 but without constants. This immediately also gives an insight into how we will later on combine finite model theory and relational algebra. In both cases, we simply define what the structure will be of the things we will be querying, not the contents.

As an example, say that we wished to define a database that defines some type simple book catalog for a book store. We would then define a database schema with two relations. The first relation symbol $Books : 4$ with an arity of four, will contain data related the books. In its four columns it will have the book title, the name of the author, the type of the book and the name of the publisher, in that order. The other relation symbol is $Publishers : 3$ with an arity of three and will contain data related to the publisher. It will have columns for the name of the publisher, its website and the day of the week that shipments from this publisher arrive, in that order. Notice how we only defined the amount of columns and the meaning of each column but not the actual value. Also notice how we had to state that the columns were in a specific order. This is because we will refer to the columns by their order and not by their name. This approach is equivalent to referring to the columns by their name, but will become useful when we expand the RA, as stated earlier.

The next concept to introduce is the concept of a *universe*. This will be the same as in Definition 2.1.2 if we assume our database schema to be a vocabulary without any constants. Of course a universe does not make any sense without a structure or, in the case of the relational algebra, a *database instance*. The database instance assigns the actual values to the relation symbols from the universe.

**Definition 4.2.2.** Assume that $D$ is a database schema and $U$ is a set of values called the *universe*. A *database instance* $I$ over $D$ is a function that maps every relation symbol $R_1 : a_1$ of $D$ to a subset $U^{a_1}$.

In this definition we refer to a relation $U^{a_1}$ where $a_1$ is some natural number and $U$ a set of values. In this situation $U^{a_1}$ means the set of all tuples that are all combinations of length $a_i$ of the values of $U$. As an example if we have $U = \{1, 2\}$, then we have $U^2 = \{(1, 1), (1, 2), (2, 1), (2, 2)\}$. Thus, we can conclude that a database instance is simply maps each relation symbol to a set of tuples of the length of the arity of that symbol that only contain values from the universe.

As an example of a database instance we will show the result of the mapping on each relation symbol and the universe. This is given by the tables in Table 4.1. Again notice how we do not give any names to the columns and simply present them in their order. In this table representation each row represents a single tuple in the set of tuples that the instance $I$ maps to a particular relation symbol. Note that $U$ is not a relationship that can be accessed, but it has been added here to show what the universe looks like in this example. In this example the universe $U$ also perfectly matches with all the values found in the relations. This collection of all values found in the relations is called the *active domain*. However, it is not required that the universe and the active domain are the same. The active domain must be a subset of the universe, but the universe could contain more. For example our universe $U$ here could have also contained the values 'THU', 'FRI', 'SAT' and 'SUN', even though we do not use them, and still be a valid universe.

### 4.2.2 Operators

Now that we have defined what the relational algebra will be operating on, it is time to define how it will operate. In the relational algebra there are two types of symbols used: *relation symbols* and *operators*. A relation symbol will return the table of the relation it refers to. This means that it will return a list of the tuples given by the list returned by the instance for that relation symbol. All the different operators allow us to then manipulate this table.

As an example take our instance of Table 4.1. If we would then write the formula $Publishers$ we would get the list of rows {(SERN Publishing, sern-pub.org, MON), (Shōnen Ace, web-ace.jp, TUE), (Instant Publisher, instant-publisher.com, MON), (Alfred A. Knopf, knopfdoubleday.com, WED)}. This collection of rows is then also called a table.

There is also a third kind of symbol that can be used. That symbol is constant relation. This is not so much a symbol type on its own, but a special kind of relation symbol where we create a relation from a single value of the universe. This is called a constant expression. We can write a constant expression as follows $\{(x)\}$ with $x$ some value from the universe. This means at it will create a relation of arity one with one row. As an example, take the universe from

| *Books* | | | |
|---|---|---|---|
| Theory of time travel | Dr. Nakabachi | Research | SERN Publishing |
| Helvetica Standard | Keiichi Arawi | Comic | Shōnen Ace |
| A Time Traveler's Tale | John Titor | Biography | Instant Publisher |
| Eragon | Christopher P. | Fantasy | Alfred A. Knopf |
| Brisingr | Christopher P. | Fantasy | Alfred A. Knopf |
| Dead Man Wonderland | Jinsei Kataoka | Manga | Shōnen Ace |

| *Publishers* | | |
|---|---|---|
| SERN Publishing | sernpub.org | MON |
| Shōnen Ace | web-ace.jp | TUE |
| Instant Publisher | instantpublisher.com | MON |
| Alfred A. Knopf | knopfdoubleday.com | WED |

| *U* |
|---|
| Theory of time travel |
| Helvetica Standard |
| A Time Traveler's Tale |
| Eragon |
| Brisingr |
| Dead Man Wonderland |
| Dr. Nakabachi |
| Keiichi Arawi |
| John Titor |
| Christopher P. |
| Jinsei Kataoka |
| Research |
| Comic |
| Biography |
| Fantasy |
| Manga |
| SERN Publishing |
| Shōnen Ace |
| Instant Publisher |
| Alfred A. Knopf |
| sernpub.org |
| web-ace.jp |
| instantpublisher.com |
| knopfdoubleday.com |
| TUE |
| MON |
| WED |

Table 4.1: An example of a database instance

| | | | |
|---|---|---|---|
| SERN Publishing | sernpub.org | MON | MON |
| Shōnen Ace | web-ace.jp | TUE | MON |
| Instant Publisher | instantpublisher.com | MON | MON |
| Alfred A. Knopf | knopfdoubleday.com | WED | MON |

Table 4.2: The result of $Publisher \times \{(MON)\}$

Table 4.1. We could create a constant relation of the value 'Eragon' by writing $\{(Eragon)\}$. This will then return a table that is exactly $\{(Eragon)\}$.

### Cartesian product

The first operator we will describe from the relational algebra is the 'Cartesian product'. This operator combines the outputs of two relational algebra expression by making all possible combinations of their rows. The Cartesian product is written as $F \times G$ for some relational algebra expression $F$ and $G$. How it works is it creates a new table to store the output in. Then for each row of $F$ it goes through every row of $G$ and puts in the output table that row that has the columns of that row of $F$ followed by the columns of the current row of $G$. Assume that the output of $F$ is a relation of arity $n$ and the output of $G$ is a relation of arity $m$. The output of $F \times G$ is then a relation of arity $n + m$.

As an example of the Cartesian product consider the database described in Table 4.1. On this instance we shall execute the query $Publisher \times \{(MON)\}$. This relation will for each row of the relation $Publisher$ create a new row for each row in $\{(MON)\}$ and add all its columns to the row from $Publisher$. However, since $\{(MON)\}$ only has a single row and a single column, a single column gets added to each row of $Publisher$. The arity of the resulting expression is $3 + 1 = 4$.

### Selection

The next operator we will discuss is the 'selection' operator. This operator takes a single table as its input and produces as its output all rows of the input table for which some condition holds. In these conditions we can use equality and references to columns. If we wish to enforce multiple filters, we simply apply another selection filter the output of the selection. This operator can be written as $\sigma_{i=j}(F)$, where $i, j \in \{1, \ldots, n\}$ if the input relation $F$ is of arity $n$. The input relation $F$ can be any kind of relational algebra expression, including of course any of the previously discussed relation symbols. Also notice how, since it only filters rows, the selection operator does not change the arity of the input relation compared to the output relation.

As an example, Assume we want to retrieve all books that are Fantasy books from our relations from Table 4.1. We would write this as $\sigma_{3=5}(Books \times$

| | | | | |
|---|---|---|---|---|
| Eragon | Christopher P. | Fantasy | Alfred A. Knopf | Fantasy |
| Brisingr | Christopher P. | Fantasy | Alfred A. Knopf | Fantasy |

Table 4.3: The result of $\sigma_{3=5}(Books \times \{(Fantasy)\})$

| | |
|---|---|
| Theory of time travel | MON |
| Helvetica Standard | TUE |
| A Time Traveler's Tale | MON |
| Eragon | WED |
| Brisingr | WED |
| Dead Man Wonderland | TUE |

Table 4.4: The result of $\pi_{1,7}(\sigma_{4=5}(Books \times Publisher))$

$\{(Fantasy)\}$). What is happening here is we first add a new column to the relation of Books with the value 'Fantasy' for each row. Then we check if the book type column and the new column are the same value. Since this value is always 'Fantasy', it will keep only those rows whose book type is 'Fantasy'. The resulting table can be found in Table 4.3. This technique of adding a new column with a constant value and then selecting based on that constant value, is how it is possible to filter on some constant value.

### Projection

Another operator from the relational algebra is the 'projection' operator. This operator can remove certain columns from the input relation. It does this by stating which columns it should keep. This written as $\pi_{i_1,\ldots,i_k}(F)$ where $i_1, \ldots, i_k \in \{1, \ldots, n\}$ for a given relational algebra expression $F$ with an arity of $n$. What this operator then does is for each row of $F$, it adds to the output table the row with only the columns listed in $i_1, \ldots, i_k$. Thus, when we project a relation to only the columns of $i_1, \ldots, i_k$ we reduce it to an arity of $k$.

As an example take the database instance of Table 4.1. If we asked for the days on which a certain book could be delivered we would do this with $\sigma_{4=5}(Books \times Publisher)$. However, we would still have two columns with the names of the publishers and a lot of other info that we do not need. To solve this we could project only those columns that are of interest to us. If we only want to keep the column with the book title and the weekly delivery date we would write $\pi_{1,7}(\sigma_{4=5}(Books \times Publisher))$. This then reduces the arity to 2, since only two columns are kept. The resulting relation is the one in Table 4.4.

| Shōnen Ace |
| --- |
| Alfred A. Knopf |

Table 4.5: The result of $(\pi_4(\sigma_{3=5}(Books \times \{(Fantasy)\})) \cup \pi_4(\sigma_{3=5}(Books \times \{(Comic)\}))) - \pi_1(\sigma_{3=4}(Publishers \times \{(MON)\}))$

**Union and difference**

The final operators left to describe for the relational algebra are the 'union' and the 'difference' operators. These operators take two relational algebra expressions of the same arity and produce the union or the difference between their rows. These are written as $F \cup G$ and $F - G$ respectively, with $F$ and $G$ two relational algebra expressions of the same arity. The reason they must be of the same arity is because if this were not the case we would get a table where the rows are sometimes of one arity and other times of a different arity. We can also write the intersection of $F$ and $G$ with only the union and difference by writing $F - (F - G)$ instead. What we do here is we remove from $F$ all the rows that occur in $F$ but not in $G$.

As an example assume again the database instance described in Table 4.1. Now assume that the query we want to ask is to get all publishers that have published a Comic or Fantasy book but not those that do their deliveries on Mondays. We would do this with the following relational algebra query:

$$\big(\pi_4(\sigma_{3=5}(Books \times \{(Fantasy)\})) \cup \pi_4(\sigma_{3=5}(Books \times \{(Comic)\}))\big) - \pi_1(\sigma_{3=4}(Publishers \times \{(MON)\}))$$

This query first takes all books that are fantasy books and keeps only their publishers. Then it does the same for comic books and takes the union of those two. Finally, it calculates the set of all publishers that deliver on a Monday, keeps only their names and then removes them from the previous union. The result of this query can be found in Table 4.5. Note that we do not have to worry about duplicate values of publishers since we take the union of the sets of rows. Set cannot hold the same value twice, thus we do not need to worry about duplicate values. This way of thinking about relational algebra expressions is called *set semantics*.

### 4.2.3 Comparison to SQL

Now that we have described the relational algebra, we need to consider how it relates to SQL. In the beginning of the section on relational algebra we claimed it to be the foundation of SQL. Yet so far, the relational algebra does not seem to bear much resemblance to SQL. However, the relational algebra expresses exactly the same queries as SQL using only queries using only the SELECT, FROM and WHERE clauses [CG85]. This includes using subqueries. Of course

if we use arithmetic in the WHERE clause we would also need to introduce arithmetic to our selection operators. In fact, one of the most powerful and useful aspects of SQL is that it can be translated to the relational algebra operators. These can then optimized and executed on an actual database. Of course, since we know that not the entirety of SQL can be expressed in the relational algebra, we will need to extend it to allow for certain operations. These operators are mostly grouping operators and aggregate functions. There are some aggregate functions however that can be expressed However, the core of querying in SQL is already present in the relational algebra. That is the reason it is called the foundation of SQL.

## 4.3 Relational calculus

Now that we have defined the relational algebra and shown that it is equal to a subset of SQL, we might ask what it can and cannot express. We could attempt to do this by creating theorems for the relational algebra that guarantee a formula must exist or that allow us to arrive at some contradiction. However, these theorems already exist in an adjacent field, namely finite model theory. In Section 2.2 we discussed one type of game that easily allows us to arrive at a contradiction to prove that something cannot be expressed in first-order logic. However, this of course only works for FO logic. This is what the *relational calculus*, sometimes shortened to *RC*, is used for: to provide a translation between relational algebra and first-order logic. The relational calculus is in fact nothing more than first-order logic applied over a vocabulary that only contains relation symbols. However, there is one difference: in the relational calculus we are allowed to introduce any data value as a constant in the formula.

We have already previously defined the first order logic in Section 2.1.4. The relational algebra is the same but in all locations where a constant symbol is permitted, a constant data value is also permitted. This difference can be overcome if we consider a structure with the same relation symbols as our database schema and an additional constant symbol for each constant data value in the formula. These constant symbols will then get the values of the data values they are replacing. This also does not prove a problem when playing an EF-game, since we get to choose the structure after the formula has been chosen. This means that the formula will fix the vocabulary, but the structure itself can still be chosen. We can then win the $n$-round EF-game over a structure with $k$ constants in it by making the structure that would win an $n + k$-round EF-game if there were no constants. Essentially we assume that the $k$ constants are the first $k$ moves in the $n + k$-round EF-game.

As an example of a relational calculus query take the relational algebra query from Table 4.5 $(\pi_4(\sigma_{3=5}(Books \times \{(Fantasy)\})) \cup \pi_4(\sigma_{3=5}(Books \times \{(Comic)\})))$ $-\pi_1(\sigma_{3=4}(Publishers \times \{(MON)\}))$. If we wish to translate this query we would start by first defining the free variables for the query and its name. Assume we

call the query $\varphi_{CFNotMon}$ and we gave it one free variable *pub*. We would then write this formula as $\varphi_{CFNotMon}(pub)$. The formula itself would be broken into three parts: one for each type of book and then one to remove the publishers that deliver on Mondays. The first part to select only publishers that have produced 'Fantasy' books can be written as $\exists a_1, a_2(Books(a_1, a_2, \text{'Fantasy}', pub))$. Similarly, we can get all 'Comic' books with the formula $\exists b_1, b_2(Books(b_1, b_2, \text{'Comic}', pub))$. With the final part we would enforce that the publisher does not deliver its books on a Monday with the following formula: $\neg\exists d(Publishers(pub, d, \text{'MON}'))$. Finally, if we combine all these parts we get the following relational calculus formula:

$$\begin{aligned}\varphi_{CFNotMon}(pub) \equiv &\big(\exists a_1, a_2(Books(a_1, a_2, \text{'Fantasy}', pub) \vee \\ &\exists b_1, b_2(Books(b_1, b_2, \text{'Comic}', pub))\big) \wedge \\ &\neg\exists d(Publishers(pub, d, \text{'MON}'))\end{aligned}$$

In this formula we combined the first two parts with a disjunction, since either may hold and then combined that with the last part with a conjunction, enforcing both must hold. In our translation of the query there are two things to notice. Firstly notice how in the relational calculus we do not need to introduce a constant relation and then equate two columns. We can simply use the constant value directly when referencing the relation. A second thing of note is how we project columns of a relation. Notice how for each column that we do not care about, we put a variable that we have existentially quantified or a constant. On the other hand if we would want to keep the column that has a constant value, we can put a variable, say $a_3$, there and then add $\wedge = const$ for some constant value *const*.

### 4.3.1 Safe relational calculus

In the previous section we showed an example of the relational algebra being translated into the relational calculus. This goes further than this one example: we can translate any relational algebra expression. This means that everything that can be expressed in the relational algebra can be expressed in the relational calculus. However, the reverse is not true. There are relational calculus formulas that cannot be expressed in the relational algebra. As an example, take the query $\varphi(x) \equiv \exists y(x = y)$. This query in the relational calculus will return the entire universe, since that is the set of all the values it will check for $x$. If this universe exactly matches the active domain, we could write this query in the relational algebra. We would do this by for each column of each relation symbol creating a projection that keeps only that column. Then we would take the disjunction of all these projections and that would be our active domain. Note that we do not have to worry about duplicate values in our active domain since we use set semantics. If however our universe is larger than the active domain, then we could never get the same set of values returned. This is because the relational algebra can only access the values from the relations and constants. This can be solved if we simply assume that we only evaluate a relational calculus

expression over the active domain. We call this way of evaluating the relational calculus using *active domain semantics*.

When using active domain semantics, we simply change the way we interpret the relational calculus. However, this no longer allows us to use the relational calculus and proofs about first-order logic to prove that things cannot be expressed. This is because in first-order logic we are still allowed to use any value from the universe when quantifying a variable. Thus, the mapping between the two that we introduced earlier is broken. A more convenient approach would be to change these queries so that they are guaranteed to only use the active domain when giving a result. The idea is to change them in such a way that they also still quantify variables over the entire universe. This would turn the relational calculus into a subset of first-order logic. However, this does mean that we can use the proof that something cannot be expressed in first-order logic, to mean that it cannot be expressed in this relational calculus. The issue is that we have not yet defined a way to identify clearly when a formula is and is not of this kind. For that we introduce the *safe relational calculus*, which allows us to syntactically determine these kinds of relational calculus expressions. The real power of the safe relational calculus is that it is exactly as powerful as the relational algebra [C$^+$72] [Ull88]. That means that the safe relational calculus and the relational algebra can express the exact same set of queries. This means that they also have the exact same limitations. Our definition here of the safe relational calculus is based on the definition given by Ullman [Ull88].

**Definition 4.3.1.** Assume that the universe of the constant data values is $V$. A relational calculus formula is *safe* if:

- It does not contain $\forall$
- Any subformula of the form $\varphi \vee \psi$ is such that $\varphi$ and $\psi$ have the same set of free variables.
- For every maximal subformula of the form $\delta_1 \wedge \cdots \wedge \delta_n$ every free variable is limited. A variable $x$ is limited if:
  - $x$ occurs free in one of the $\delta$'s that is not negated and that is not of the form $x = y$ or $y = x$
  - one of the $\delta$'s is of the form $x = v$ or $v = x$ where $v \in V$
  - one of the $\delta$'s is of the form $x = y$ or $y = x$ and $y$ is already limited

Notice how this definition uses the concept of a maximal subformula of a given form. That means we take all subformulas of a certain form and try to expand them until we can no longer expand then without them losing their form. As an example, say we try to find the maximal subformula of the form $\delta_1 \wedge \cdots \wedge \delta_n$. Assume the formula we are looking for the maximal subformulas is $(a \wedge b) \vee c \vee (b \wedge (c \vee d \vee (b \wedge e)))$. The maximal subformulas we would find would then be $a \wedge b$, $c$ twice, $(b \wedge (c \vee d \vee (b \wedge e)))$, $d$, $b \wedge e$. Notice how we have

multiple maximal subformulas of that are a subformula of a different maximal subformula. This is because by not being able to expand a formula $\delta$ we mean that there is no other subformula of the form $\delta \wedge \sigma$ or $\sigma \wedge \delta$.

## 4.4 Relational meta-algebra

Now that the relational algebra and the relational calculus have been defined, we can start to add features on top of these. First we shall start by expanding the relational algebra with additional operators and additional concepts. The objective of these operators is, similar to the additions in Meta-SQL, to allow us to query, modify and execute stored queries. In Section 4.5 we shall then translate these additional operators into an extension of the relational calculus. However, before introducing these new operators, it is important that we introduce some new concepts that allow us to define the behavior of these operators. Both this section and Section 4.5 are based upon the work of Van den Bussche et al. who defined both the relational meta-algebra and the relational meta-calculus [NVVV99].

### 4.4.1 Types

The first new concept we will introduce is the concept of *types*. A type is a tuple $\tau = [\tau_1, \ldots, \tau_n]$ describing the columns of a relation where each element is either 0 or $\langle m \rangle$ where $m$ is a natural number. If the value is 0, we say that column is a *data column*. If the value is of the form $\langle m \rangle$, we say that the column is an *expression column*.

**Definition 4.4.1.** Let $S$ a database schema, $\tau = [\tau_1, \ldots, \tau_n]$ be a type and $U$ to be the universe over which the tuple will be defined. A *tuple of type $\tau$ over* $S$ is a tuple $(x_1, \ldots, x_n)$, such that for each $1 \leq i \leq n$:

- if $\tau_i$ is 0, then $x_i$ is a data value, which means $x_i \in U$
- if $\tau_i$ is $\langle m \rangle$, then $x_i$ is a relational algebra expression over $S$ of arity $m$

A *relation of type $\tau$ over* $S$ is a set of tuples of $\tau$ over $S$.

As an example, assume a database schema $S = \langle R, S \rangle$ with $R : 2$ and $S : 3$. The tuple $(1, 5, R, R \times S, \text{`hello}')$ would have the type $\tau = [0, 0, \langle 2 \rangle, \langle 5 \rangle, 0]$. Also notice how a type of purely 0's is a normal relation in a database schema. In the previous example the types of $R$ and $S$ would be $[0, 0]$ and $[0, 0, 0]$ respectively.

### 4.4.2 Meta-level schema

Now that we have defined a type, we clearly need to define a new type of relations and schema. The database schema from the relational algebra, contains relations that only contain data vales and whose types are of the form $[0, \ldots, 0]$. We call such a schema the *object-level schema*. However, as we have seen with types, we

| *Books* | | | |
|---|---|---|---|
| Theory of time travel | Dr. Nakabachi | Research | SERN Publishing |
| Helvetica Standard | Keiichi Arawi | Comic | Shōnen Ace |
| A Time Traveler's Tale | John Titor | Biography | Instant Publisher |
| Eragon | Christopher P. | Fantasy | Alfred A. Knopf |
| Brisingr | Christopher P. | Fantasy | Alfred A. Knopf |
| Dead Man Wonderland | Jinsei Kataoka | Manga | Shōnen Ace |

| *Publishers* | | |
|---|---|---|
| SERN Publishing | sernpub.org | MON |
| Shōnen Ace | web-ace.jp | TUE |
| Instant Publisher | instantpublisher.com | MON |
| Alfred A. Knopf | knopfdoubleday.com | WED |

| *Triples* | |
|---|---|
| FantasyBooks | $\pi_{1,2,4}(\sigma_{3=5}(Books \times \{(Fantasy)\}))$ |
| BookDates | $\pi_{1,2,7}(\sigma_{4=5}(Books \times Publishers))$ |
| AuthorBookPairs | $\pi_{1,5,6}(\sigma_{2=6}(Books \times Books))$ |
| Publishers | $Publishers$ |

Table 4.6: An example of a combined instance

can also have relations that contain relational algebra expressions together with data values. We call a schema with relations of this type a *meta-level schema*.

**Definition 4.4.2.** A *meta-level schema* $M$ is a finite set of relation names, where each relation name has an associated type. If a relation $R$ has type $\tau$ we write this as $R : \tau$. Take an object-level schema $S$ disjoint from $M$. An *instance of* $M$ *over* $S$ is a mapping $J$ on $M$ which assigns to each relation name $R : \tau \in M$ a relation of type $\tau$ over $S$. The pair $(S, M)$ is called a *combined schema*. An instance over a combined schema $(S, M)$ is the union of an instance of $S$ and an instance of $M$ over $S$ and is called a *combined instance*.

As an example of a combined instance, assume we have an object-level schema $S = \langle Books, Publishers \rangle$ and a meta-level schema $M = \langle Triples \rangle$. Assume as well that $Triples : [0, \langle 3 \rangle]$. Then the relations shown in Table 4.6 is a combined instance of $(S, M)$.

### 4.4.3 Operators

We will now discuss the new operators and give examples of each of them. However, before doing so we need to introduce the concept or rewrite rules used by the rewrite operators. On top of this we also need to show how the operators

of the relational algebra have changed with the introduction of a combined schema and types.

### Changed RA Operators

To start our description of operators, we will first discuss how the operators of the relational algebra have changed. They have not changed in the way that they calculate their results. However, the operators have changed with regard to on what relations they can operate and what types they produce. Assume a universe $U$, an object-level schema $S$, a meta-level schema $M$. The following list will shall give all the changes to the relational algebra operators when they defined over a combined schema $(S, M)$.

- The constant expression becomes $\{(v)\} : [0]$ with $v \in U$
- The relation symbol $R : n \in S$ becomes an expression $R : [0, \ldots, 0]$ with $n$ zeros.
- The relation symbol $R : \tau \in M$ becomes an expression $R : \tau$
- The union and difference of the expressions $e_1 : \tau$ and $e_2 : \tau$ become $(e_1 \cup e_2) : \tau$ and $(e_1 - e_2) : \tau$ respectively
- The Cartesian product of the expression $e_1 : \tau$ and $e_2 : \omega$ with $\tau = [\tau_1, \ldots, \tau_n]$ and $\omega = [\omega_1, \ldots, \omega_m]$ becomes $(e_1 \times e_2) : [\tau_1, \ldots, \tau_n, \omega_1, \ldots, \omega_m]$
- The selection of columns $i$ and $j$ being the same for an expression $e : \tau$ with $\tau = [\tau_1, \ldots, \tau_n]$ becomes $\sigma_{i=j}(e) : \tau$ where $i, j \in \{1, \ldots, n\}$ and $\tau_i = \tau_j$
- The projection of a set of columns $\{i_1, \ldots, i_p\}$ of an expression $e : \tau$ with $\tau = [\tau_1, \ldots, \tau_n]$ becomes $\pi_{i_1,\ldots,i_p}(e) : [\tau_{i_1} \ \ldots, \tau_{i_p}]$

A first thing to note is, how the first two changes merely replace the arity of a relation symbol with its type. The next change is the reason this has to happen: for a meta-level relation, arity is no longer a sufficient description. This change thus allows us to refer to a relation from the meta-level schema using the relational algebra operators. The next set of changes are to the union and difference operators. Again for the same reason we enforced that the operands need to have the same arity, it makes sense to enforce that they have the same type. If this were not that case, we would have certain rows of one type and other rows of a different type. But this would not allow us to give a single type for the result of the expression.

The next change is to the Cartesian product, where we simply follow the same logic as when defining it for the relational algebra. In the relational algebra we said we simply added new columns to the end of the columns of the first operand. With this change we do the exact same, but now we allow these columns to contain relational algebra expressions thanks to using types rather

than arity. The changes made to the selection are because comparing columns of a different type makes little sense as they will never be equal. Finally, the changes to the projection are just so that the output type matches what would happen if we applied it to an expression with a given type. Since we only keep a specific set of columns, we also only keep the types of those columns in the type of the output relation.

### Rewrite rules

Now that the changes to the relational algebra operators have been shown, we shall introduce a concept used in one of the new operators: *rewrite rules*. A rewrite rule is a pair of expressions where an occurrence of the first expression will be replaced by the second expression. However, rather than simply be relational algebra expression, these can contain extra symbols in them that will be replaced with the RA expression in a column when executed.

**Definition 4.4.3.** Let $S$ be an object-level schema, $\tau = [\tau_1, \ldots, \tau_n]$ be a type. Let $C \subseteq \{1, \ldots, n\}$ be the set of expression columns in $\tau$ and let for each $j \in C$ $\tau_j = \langle l_j \rangle$. A *rewrite rule over $S$ with respect to $\tau$* is a rule of the form $\alpha \to \beta$ where $\alpha$ and $\beta$ are RA expressions of the same arity over $S \cup \{\square_j \mid j \in C\}$. Each $\square_j$ is an *expression variable of arity $l_j$*. $\alpha$ and $\beta$ are called *patterns with respect to $\tau$*.

Notice how we called $\square_j$'s expression variables. This is because these are not expressions in their own regard, but are expressions that are a stand in for a value that will be given by a row. This is very much like a variable, which is a stand in for a value until it at some point is assigned one. Here $\square_j$ will be assigned the value of column $j$ for each row. $j$ is always an expression column because it is an element of $C$, which is the set of all expression columns in $\tau$.

As an example say we have a schema $S = \langle R : 4, S : 2, T : 2 \rangle$ and a type $\tau = [0, \langle 2 \rangle]$. A rewrite rules over $\tau$ would for example be $(S \times \square_2) \to R$. Notice how $S \times \square_2$ and $R$ are both of arity 4 since $\square_2$ is of arity 2. Assume the rows to which this is being applied are $\{(1, S), (5, T - S)\}$. The rewrite rule for the first row then becomes $(S \times S) \to R$. For the second row the rewrite rule would become $(S \times (T - S)) \to R$.

### Rewrite operators

The first new operators we will introduce are the *rewrite operators*. These operators take as input a column number $i$ which is an expression column, a rewrite rule $\alpha \to \beta$ and apply this rewrite rule to column of the given RMA expression $e$. There are two rewrite operators. The first is the 'rewrite-all' operator which replaces every occurrence of the pattern $\alpha$ with the pattern $\beta$. The operator then adds a new column to the end and places the rewritten formula in this column for the row. For the given RMA expression, column and rewrite rule this can be written as $\mathit{rewrite\text{-}all}_{i:\alpha\to\beta}(e)$.

| | | |
|---|---|---|
| FantasyBooks | $\pi_{1,2,4}(\sigma_{3=5}(B \times \{(F)\}))$ | $\pi_{1,2,4}(\sigma_{3=5}(B2 \times \{(F)\}))$ |
| BookDates | $\pi_{1,2,7}(\sigma_{4=5}(B \times P))$ | $\pi_{1,2,7}(\sigma_{4=5}(B2 \times P))$ |
| AuthorBookPairs | $\pi_{1,5,6}(\sigma_{2=6}(B \times B))$ | $\pi_{1,5,6}(\sigma_{2=6}(B2 \times B2))$ |
| Publishers | $P$ | $P$ |

Table 4.7: Results of the query $\textit{rewrite-all}_{2:(B)\to(B2)}(\mathit{Triples})$

| | | |
|---|---|---|
| FantasyBooks | $\pi_{1,2,4}(\sigma_{3=5}(B \times \{(F)\}))$ | $\pi_{1,2,4}(\sigma_{3=5}(B2 \times \{(F)\}))$ |
| BookDates | $\pi_{1,2,7}(\sigma_{4=5}(B \times P))$ | $\pi_{1,2,7}(\sigma_{4=5}(B2 \times P))$ |
| AuthorBookPairs | $\pi_{1,5,6}(\sigma_{2=6}(B \times B))$ | $\pi_{1,5,6}(\sigma_{2=6}(B \times B2))$ |
| AuthorBookPairs | $\pi_{1,5,6}(\sigma_{2=6}(B \times B))$ | $\pi_{1,5,6}(\sigma_{2=6}(B2 \times B))$ |

Table 4.8: Results of the query $\textit{rewrite-one}_{2:(B)\to(B2)}(\mathit{Triples})$

The second rewrite operators is the 'rewrite-one' operator. This operator checks all the locations of the pattern $\alpha$. For each of these locations it generates a new expression which has that one occurrence of $\alpha$ replaced and the rest of the expression left unchanged. A new column is then added and a row is created for each of the generated formulas. Another way to think of this is that we take a big union of a collection of Cartesian products. Each of the Cartesian products is then the product of a single row and the formulas it generated. If we apply this operator with the given expression, column number and rewrite rule we would write $\textit{rewrite-one}_{i:\alpha\to\beta}(e)$.

For both of these operators, assume the input is of a type $\tau = [\tau_1, \ldots, \tau_n]$ and the column is $i$. Then regardless of the arity of the two patterns in the rewrite rule the type of the output will be $\tau = [\tau_1, \ldots, \tau_n, \tau_i]$.

As an example of this operator, take the instance described in Table 4.6. Assume that $B2$ is a new relation symbol of the object-level schema and that $B$ stands for *Books*, $P$ for *Publishers* and $F$ for *Fantasy*. If we then executed the query $\textit{rewrite-all}_{2:(B)\to(B2)}(\mathit{Triples})$ we get the results in Table 4.7. If we execute the query $\textit{rewrite-one}_{2:(B)\to(B2)}(\mathit{Triples})$, we would get the results in Table 4.8. Notice how the rewrite-one query created multiple rows for the row with 'AuthorBookPairs' but had no row for 'Publishers'. This is because there were multiple locations of 'Books' to replace in the row with 'AuthorBookPairs' and none to replace in the row with 'Publishers'. However, the rewrite-all query kept the row with 'Publishers' because if a pattern does not occur in an expression, the expression with all those patterns replaced is the same expression.

**Extract**

The next operator we will discuss is the *extract* operator. This operator lets us extract all subformulas of a given arity $m$ from a given column $i$, which is an expression column. Then for each extracted sub expression of the given

| | | |
|---|---|---|
| FantasyBooks | $\pi_{1,2,4}(\sigma_{3=5}(B \times \{(F)\}))$ | $\pi_{1,2,4}(\sigma_{3=5}(B \times \{(F)\}))$ |
| BookDates | $\pi_{1,2,7}(\sigma_{4=5}(B \times P))$ | $\pi_{1,2,7}(\sigma_{4=5}(B \times P))$ |
| BookDates | $\pi_{1,2,7}(\sigma_{4=5}(B \times P))$ | $P$ |
| AuthorBookPairs | $\pi_{1,5,6}(\sigma_{2=6}(B \times B))$ | $\pi_{1,5,6}(\sigma_{2=6}(B \times B))$ |
| Publishers | $P$ | $P$ |

Table 4.9: Results of the query $extract_{2:3}(Triples)$

| | | |
|---|---|---|
| FantasyBooks | $\pi_{1,2,4}(\sigma_{3=5}(B \times \{(F)\}))$ | $\{(FantasyBooks)\}$ |
| BookDates | $\pi_{1,2,7}(\sigma_{4=5}(B \times P))$ | $\{(BookDates)\}$ |
| AuthorBookPairs | $\pi_{1,5,6}(\sigma_{2=6}(B \times B))$ | $\{(AuthorBookPairs)\}$ |
| Publishers | $P$ | $\{(Publishers)\}$ |

Table 4.10: Results of the query $wrap_1(Triples)$

arity for this row, it creates a new row which adds a column with the extracted subformula onto the row. This is very similar to how the rewrite-one works for each occurrence of the pattern to be replaced, instead of subformula to be extracted. If the RMA expression this is executed on is $e$, then this is written as $extract_{i:m}(e)$.

To explain this operator better consider let us consider an example. Take the instance described in Table 4.6. Assume that $B$ stands for $Books$, $P$ for $Publishers$ and $F$ for $Fantasy$. If we then execute the query $extract_{2:3}(Triples)$ we get the results in Table 4.9. Notice how there are two rows for 'BookDates' since it contains two different subformulas that have an arity of 3

### Wrap

The next operator that the relational meta algebra introduces is the *wrap* operator. This operator allows us to introduce data values as a constant unary relation into a new expression column. Assume that column $i$ is a data column of some relational meta-algebra expression $e$. The operator would then for each row take the data value stored in column $i$, let us say value $x$, and add a new column with the expression $\{(x)\}$.

As an example take the instance of Table 4.6 again. Assume that $B$ stands for $Books$, $P$ for $Publishers$ and $F$ for $Fantasy$. If we execute the query $wrap_1(Triples)$, we get the result shown in Table 4.10. Notice how the last column still only contains a relational algebra expression and not a relational meta-algebra expression. This is because the values within the constant relations are data value, in this case strings, and not references to a relation. However, to prevent confusion in the case of strings, it is typically better to quote these data values.

| | | | | |
|---|---|---|---|---|
| Fantasy Books | $\pi_{1,2,4}(\sigma_{3=5}(B \times \{(F)\}))$ | Eragon | Christopher P. | Alfred A. Knopf |
| Fantasy Books | $\pi_{1,2,4}(\sigma_{3=5}(B \times \{(F)\}))$ | Brisingr | Christopher P. | Alfred A. Knopf |
| BookDates | $\pi_{1,2,7}(\sigma_{4=5}(B \times P))$ | Theory of time travel | Dr. Nakabachi | MON |
| BookDates | $\pi_{1,2,7}(\sigma_{4=5}(B \times P))$ | Helvetica Standard | Keiichi Arawi | TUE |
| BookDates | $\pi_{1,2,7}(\sigma_{4=5}(B \times P))$ | A Time Traveler's Tale | John Titor | MON |
| BookDates | $\pi_{1,2,7}(\sigma_{4=5}(B \times P))$ | Eragon | Christopher P. | WED |
| BookDates | $\pi_{1,2,7}(\sigma_{4=5}(B \times P))$ | Brisingr | Christopher P. | WED |
| BookDates | $\pi_{1,2,7}(\sigma_{4=5}(B \times P))$ | Dead Man Wonderland | Jinsei Kataoka | TUE |
| Author BookPairs | $\pi_{1,5,6}(\sigma_{2=6}(B \times B))$ | Eragon | Brisingr | Christopher P. |
| Author BookPairs | $\pi_{1,5,6}(\sigma_{2=6}(B \times B))$ | Brisingr | Eragon | Christopher P. |
| Publishers | $P$ | SERN Publishing | sernpub.org | MON |
| Publishers | $P$ | Shōnen Ace | web-ace.jp | TUE |
| Publishers | $P$ | Instant Publisher | instant publisher.com | MON |
| Publishers | $P$ | Alfred A. Knopf | knopfdouble day.com | WED |

Table 4.11: Results of the query $eval_2(Triples)$

### Eval

The final operator we will add is the *evaluation* operator, often shortened to the *eval* operator. This operator takes a column $i$ which is an expression column and adds the results of evaluating that expression to the row. For a given relational meta-algebra expression $e$ this is written as $eval_i(e)$. This again works similar to the rewrite-one and the extract operator, where if there are multiple rows returned, a new row will be created for each of these returned rows, appending it to the original row. If $e$ is of type $\tau = [\tau_1, \ldots, \tau_n]$ and $\tau_i = \langle m \rangle$ then the type after the eval is $[\tau_1, \ldots, \tau_n, 0, \ldots, 0]$ with $m$ zeros being added. This is rather logical since a column with an expression of arity $m$ will cause $m$ columns to be added. All these columns will also be of type 0, since our relations can only contain RA expressions over the object-level schema.

As an example take the instance described by Table 4.6. If we execute the query $eval_2(Triples)$, we get the result given in Table 4.11. From this result it is clear how rows which have multiple rows of results for their query also have multiple rows in the output of the 'eval' operator. The 'eval' operator also clearly shows what the benefits are of using a typing system, compared to simply distinguishing between expression and data columns. Thanks to the typing system the result of the 'eval' operator is again a table. If we did not enforce the typing, we would not know when executing expressions from a given column, how many columns would be added. This in turn could then result in some rows having more columns than others, which would make the output no longer a table.

### 4.4.4 Comparison to Meta-SQL

Now that we have described the relational meta-algebra and its operators, we will discuss its relation to Meta-SQL. To start, it is very clear that the evaluation function and the 'eval' operator are essentially the same. Both operators take a query and execute it. On top of this both operators can produce multiple rows from a single input row and return them together with their original row. However, the Meta-SQL also has the UEVAL function, which allows it to evaluate queries of which we do not know the arity that they will return. The results are then stored in an XML document. However, this is not possible in the relational meta-algebra. We could abandon the typing system and only distinguish between data and expression columns. However, this would lead to issues when trying to execute the 'eval' since we do not know the amount of tables and thus cannot output a table. To solve this we could add a new operator 'ueval' which simply stores every result as a series of rows that produce the result of a single row. This could be done by producing a large Cartesian product of constant relations. However, we could never access all the columns of the rows together. We could not evaluate it and add its columns to the end of the current row for obvious reasons. We could extract all subformulas of arity one, but that would split the different columns over multiple rows. Using a 'rewrite-one' operator has similar issues. And finally the 'rewrite-all' operator could not really access all these values at the same time.

Another comparison that could be drawn, is that all newly introduced operators without 'eval' essentially fulfill the same role as XSLT did in Meta-SQL. That is to say that both are added so they can manipulate stored queries. However, the operators of the relational meta-algebra are clearly far less powerful. As an example, they could not create a column that contains a query of the relational meta algebra. XSLT on the other hand could create query in the Meta-SQL. However, the issue is that XSLT is Turing-complete, which means we can make no guarantees of it ending. The operators from the relational algebra on the other hand are guaranteed to always return an answer. This means that it is potentially easier to optimize for the relational meta algebra than it is for XSLT. However, this would require entirely novel research into

how to optimize queries from the relational algebra, where XSLT already has optimizers.

## 4.5 Relational meta-calculus

Now that the relational meta-algebra has been described, we can again attempt to translate it into an equivalent calculus. This equivalent calculus will be called the *relational meta-calculus*, which we will sometimes shorten to *RMC*. This calculus will be similar to the relational calculus, since it is the calculus counterpart of an expansion of the relational algebra. Because of this we will also start from the relational calculus and then add the different new operators. After that we will introduce a safe variant of the calculus that will allow us to prove what is and is not possible in the relational meta-algebra using techniques from finite model theory. However before we can do any of those, we need to first translate the concepts of types and rewrite rules into their RMC counterparts. Again, this entire section is based on the paper by Van den Bussche et al. that defined the relational meta-calculus [NVVV99].

### 4.5.1 Types and rewrite rules

Before we can add new operators, we need to translate the concepts introduced in the relational meta-algebra. The first of these is the type system of the relational meta-algebra. This system on its own cannot work, since in the relational calculus we simply work with a single value from a column rather than an entire row. However, we can take over the types that a certain column has and use those. As an example take, a *data variable* $x$. A data variable is a variable that contains a normal data value. This means that its value likely comes from a data column. However, the correct definition of a data variable, is a variable *of sort* 0. This means that if it were a value from a column in the relational meta-algebra, this column would have type 0. This also means that all data variables and data values are of the sort 0. Likewise, an *expression variable* $y$ which contains an expression of arity $m$ is of the sort $\langle m \rangle$. Data variables are thus achieved by equating them to others data variables or binding them to a column of a relation that is a data column. Similarly, an expression variable is achieved by equating it to a different expression variable or by binding it to a column that is an expression column. This thus means that the relation symbols in our algebra can be both symbols from the object-level schema and form the meta-level schema. It also means that we extend the equality relation to support equating expression values and variables and not only data variables and data values. We also enforce that the values and variables must be of the same sort.

The next concept we need to translate into the relational calculus, is the concept of rewrite rules. In the relational meta-algebra we described these in terms of the object-level schema $S$ augmented with some extra symbols $\square_j$ for

each column $j$ that was an expression column. In the relational calculus we do not pass around tables with columns but variables containing the data of these columns. Thus, to create rewrite rules we will define how to create the patterns. A pattern in the relational meta-calculus is an RA expression over the object-level schema $S$ augmented with a finite set of expression variables. Such an expression of arity $n$ is then a *term* of sort $\langle n \rangle$. A term is an additional new concept that we introduce that encapsulates a set of expressions that can occur anywhere a data value or data variable could in the RC. A pattern created this way will also be evaluated similar to a rewrite rule, where the expression variables within it will be filled in with the value assigned to them. This then generates the actual value of the pattern that will be looked for. On top of this, thanks to every expression variables being of a known sort, we know what the arity of the pattern will be.

### 4.5.2 New operator translations

Now that the preliminaries have been defined, we shall add translations of the new operators of the relational meta-algebra to the relational calculus. These translations will take the shape of predicates. A predicate can be seen as a relation whose set of tuples is always known. This also means that when creating proofs about not being to express certain queries we need to also consider these relations on top of the other relations. For example, in an EF-game we could consider these to be relations in the structure, but we cannot choose their contents. This allows us to still use proofs, like creating structures where the Duplicator can always win an $n$-round EF-game. This then also intuitively shows how this gives us additional power. Because there are now more relations the Duplicator must take into account, its winning strategy has to become more complex, if it can still win at all.

The first operator we will translate is the 'wrap' operator. This will be translated into a term of the form $\{(x)\}$ where $x$ is a data variable, which means $x$ is of sort 0. However, the sort of the term itself is $\langle 1 \rangle$. Creating a term this way makes sense, since we extract data values from the relations by using a data variable and the 'wrap' operator wraps values from a data column. Notice how this is a term, meaning it can appear in any place a variable could.

The next operator we will translate is the 'extract' operator. For this we will introduce a 'subexpression' relation, similar to how we always have the equality relation. It will be written as $s_1 \leq_m s_2$ for two expression variables $s_1$ of sort $\langle m \rangle$ and $s_2$ any expression variable. In this case $s_1$ is a subexpression of $s_2$ with arity as $m$. This also means that $s_1$ can have multiple values associated with it per $s_2$. Again it is clear how this naturally maps to the $extract_{m:i}$ operator if $s_2$ contains the value from column $i$.

Next we will introduce two predicates that translate the two 'rewrite' operator. These are the $\mathit{rewrite\text{-}one}(t_1, t_2, t_3, t_4)$ and $\mathit{rewrite\text{-}all}(t_1, t_2, t_3, t_4)$. If we compare these to the relational algebra expression $\mathit{rewrite\text{-}all}_{i:\alpha\to\beta}e$, then $t_1$ contains the value contained in the $i$-th column of $e$, $t_2$ represents $\alpha$, $t_3$ represents $\beta$ and $t_4$ represents the value in the newly added column in the output. This also means that $t_1$ and $t_4$ must be of the same sort and $t_2$ and $t_3$ must be of the same sort. This typing is also enforced for the relational meta-algebra expression. For the 'rewrite-one' predicate this is similar. Note however that this predicate, similar to $s_1$ in $s_1 \leq s_2$, may have multiple values it can be assigned.

The final operator for which we shall introduce a predicate is the 'eval' operator. Although, rather than introduce a single predicate, we introduce a set of predicates. For a given expression variable $t$ of sort $\langle n \rangle$ we will have the predicate $\mathit{eval}(t, x_1, \ldots, x_n)$. The values of $x_1, \ldots, x_n$ are assigned so that they form a row of the results of evaluating the expression in $t$.

Finally as an example we will translate the relational algebra expression $\pi_{4,5,6}(\mathit{eval}_3(\mathit{rewrite\text{-}one}_{2:(B)\to(B2)}(\mathit{Triples})))$ to the relational meta-calculus. This becomes the formula

$$\varphi(x_1, x_2, x_3) \equiv \exists t_1, t_2, t_3(\mathit{Triples}(t_1, t_2) \wedge \mathit{rewrite\text{-}one}(t_2, B, B2, t_3) \wedge \mathit{eval}(t_3, x_1, x_2, x_3))$$

Notice how in this formula $B$ and $B2$ are simply used as terms. This is because of course a relation name form the object-level schema on its own is of course also an RA expression over the object-level schema.

### 4.5.3 Safe Relational Meta-Calculus

Similarly to the relational calculus, if we do not put restrictions on the formulas of the relational meta-calculus it becomes more powerful than the relational meta-algebra. This then no longer allows us to prove that something can be expressed using the relational meta-calculus. Thus, in this case we also define a *safe relational meta-calculus* which is exactly equivalent to the relational meta-algebra[NVVV99]. This continues on the safe relational calculus. We will however still give the entire definition here, since it makes changes in a lot of the different parts of the safe relational calculus definition.

**Definition 4.5.1.** Assume that the universe of the constant data values is $V$. A relational meta-calculus formula is *safe* if:

- It does not contain $\forall$
- Any subformula of the form $\varphi \vee \psi$ is such that $\varphi$ and $\psi$ have the same set of free variables.

- For every maximal subformula of the form $\delta_1 \wedge \cdots \wedge \delta_n$ every free variable is limited. A data variable $x$ is limited if:
  - x occurs free in one of the $\delta$'s that is not negated and that is not of the form $x = y$, $y = x$ or *eval*
  - one of the $\delta$'s is of the form $x = v$ or $v = x$ where $v \in V$
  - one of the $\delta$'s is of the form $x = y$ or $y = x$ and $y$ is already limited
  - one of the $\delta$'s is of the form $\{(x)\} = y$ or $y = \{(x)\}$ and $y$ is already limited
  - one of the $\delta$'s is of the form $eval(t, y_1, \ldots, y_m)$ and $x$ is one of the $y$'s and all variables that occur in $t$ are limited
- An expression variable $x$ is limited if:
  - $x$ occurs free in one of the $\delta$'s that is not negated and that is not of the form $t_1 = t_2$, $t_1 \leq t_2$, *rewrite-one*, *rewrite-all* or *eval*
  - one of the $\delta$'s is of the form $t_1 = t_2$ or $t_2 = t_1$ where $x$ occurs in $t_1$ or is $t_1$ and $t_2$ or all the variables in $t_2$ are already limited
  - one of the $\delta$'s is of the form $t_1 \leq_m t_2$ where $x$ occurs in $t_1$ or is $t_1$ and $t_2$ or all the variables in $t_2$ are already limited
  - one of the $\delta$'s is of the form $\{(y)\} = x$ or $x = \{(y)\}$ and $y$ is already limited
  - one of the $\delta$'s is of the form $rewrite\text{-}all(t_1, t_2, t_3, t_4)$ or $rewrite\text{-}one(t_1, t_2, t_3, t_4)$ and $x$ occurs in $t_4$ or is $t_4$ and all variables occurring in or exactly $t_1, t_2, t_3$ are limited

Notice how for 'eval' and the 'rewrite' operators only variables that are the equivalent to the newly added output can be limited by them. This makes sense, because for the 'eval' operator, for any set of data variables an infinite amount of expressions can be created that give them as output. Similarly, for the 'rewrite' operators if any of $t_1, t_2, t_3$ are not limited, yet there are situations where an infinite set of expressions are possible for that variable. As an example say that $t_2$ is missing in a rewrite-all and $t_1$ and $t_2$ are the same and $t_3$ does not occur in either. Then $t_2$ could be any expression that is not a subexpression of $t_1$. A similar case can be made if $t_3$ is the only one not yet limited. For $t_1$ if it is not limited, it is not possible to know what occurrences of $t_3$ in $t_4$ were replacements and which ones were already there in $t_1$. Finally also note how if $x$ is in or is $t_2$ in $t_1 \leq_m t_2$ and $t_1$ is limited, then $x$ is not limited either. Obviously there is an infinite amount of expressions that a given expression can be a subexpression of arity $m$ of.

# Chapter 5

# Code Querying Approach Comparison

Previously we have seen two different approaches for how to query code: an algorithmic approach of where we write an algorithm to extract the fragments from the source code and a logical and algebraic approach where we create a language meant to manipulate and query a structure that represents code. In this chapter we will attempt to compare these different approaches by trying to represent some of the common results that wish to be obtained from the algorithmic approach, as queries in the logical and algebraic approach. We will start off by assuming the similar pairs are already given. The first type of query we will discuss is one where we attempt to simply find all patterns that occur a certain amount of times. The next query we will discuss is one where we attempt to find pairs of patterns where one pattern occurs fewer times than the other. We will prove that our current tools are not strong enough to express this query. Next we create a new language, adding additional operators specific to general code querying. While adding new operators, we will prove which subsets of this new language are strong enough to express the second query. Once we have found this set of operators we will investigate if our new language can also express the query of trying to find all similar pairs of code patterns. We will also consider how difficult it would be to translate the previously defined tree-mining algorithms and other types of similarity. Finally, we will discuss how this new language compares to other languages, such as the relational meta-algebra and the Meta-SQL.

The reason we take the approach of defining our own new language, is twofold. Firstly, none of the previously defined logical and algebraic languages can work with arbitrary AST's. Thus, we need to define a new logical language to do these operations. Secondly, defining a new language to which we add operators one by one, allows us to find out exactly which operators add the most power to a language.

## 5.1 Frequent pattern mining

When extracting code with some algorithm, one of the most common types of results that these algorithms wish to achieve, either as an end-goal or an intermediate one, is to find all patterns that are above a certain frequency. This concept is called frequent pattern mining. For this approach we will assume that there is already some relation $R(C1, C2)$ that contains all pairs of patterns that are considered similar and that there is a relation $T(C)$ that contains all the patterns. If we wanted to find all patterns that have at least $n$ other similar patterns and thus are considered frequent, this could be written in SQL as follows:

```
SELECT T.C AS c
FROM T, R
WHERE T.C = R.C1
GROUP BY T.C
HAVING COUNT(R.C2) >= n
```

Of course in this example we would have to replace $n$ with the actual value we want to use as the cut-off of what is frequent, if we wish to be able to execute this query. Note that this query uses a GROUP BY-clause and a COUNT aggregator. These types of clauses and operators are typically elements that provide problems when translating to the relational algebra or the relational calculus. Next we shall show that it is possible to express this query in the relational calculus and thus in first order logic. The relational calculus query, that we shall call $freq$, is as follows:

$$freq(n) := \{c \mid T(c) \wedge \exists c_1, \ldots, c_n \left( \left( \bigwedge_{1 \leq i < j \leq n} c_i \neq c_j \right) \wedge \left( \bigwedge_{i=1}^{n} R(c, c_i) \right) \right)\} \quad (5.1)$$

This query is built up from two parts joined together through a $\wedge$. On the left side there is $\bigwedge_{1 \leq i < j \leq n} c_i \neq c_j$ stating that all the variables created by the existential quantifier must be different. On the right side there is $\bigwedge_{i=1}^{n} R(c, c_i)$ stating that each of these variables generated must be a code fragment that is similar to $c$. This clearly shows that for any given $n$ we can create a relational calculus query, and thus also a first-order logic formula, that expresses the concept of frequency pattern mining. Note that since this is written in relational calculus it can also be expressed in relational algebra and thus also in the relational meta-algebra. However, the more important result is that it can be expressed in first order logic, meaning we have no need of the more complex tools offered by the relational meta-algebra or second order logic or some subset of it.

## 5.2 Pattern frequency comparison

Another potential query that could be asked, is if a refactoring of some piece of code has resulted in a more common pattern being used or if it has resulted in a less common pattern being used. The way this would most likely be queried, is to check for all the changes to code that made that code use a less common pattern. For this again the assumption is made that there is some relation $R(C1, C2)$ that contains all pairs of patterns that are considered similar and that there is a relation $T(C1, C2)$ where $C1$ contains the old fragment and $C2$ is the updated fragment. Written as an SQL query it would look as follows:

```
SELECT T.C1 AS old, T.C2 AS new
FROM T
WHERE ( SELECT COUNT(R.C2) AS count
        FROM R
        WHERE R.C1 = T.C1
) > ( SELECT COUNT(R.C2) AS count
        FROM R
        WHERE R.C1 = T.C2
)
```

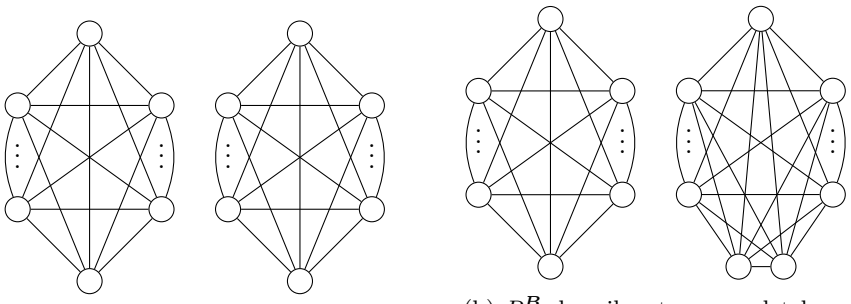
Note how again there is use of the aggregator COUNT here. Again, use of aggregators raises the question of whether it can be expressed in first-order logic. We will prove that this is not possible to be expressed in FO logic and then try to find an extension of the relational algebra where it can be expressed.

### 5.2.1 First-order logic

When trying to express this query in relational calculus, and thus in first-order logic, we could try to use the same trick used in Equation 5.1. However, this trick generates a formula that can count up to $n$ and although it would be possible to state that one has $n$ or higher and the other has strictly less than $n$, we cannot do this for every possible $n$. We cannot know what the value of $n$ will be or even give an upper bound for it, since the universe of the structure over which it will be evaluated is not yet know and cannot be limited. This means that the same trick cannot be used. In fact, we shall prove that it cannot be expressed in FO logic, and thus in relational calculus and the relational algebra, at all with an Ehrenfeucht-Fraïssé game.

To prove this, we will first need to look at how the query is fundamentally structured. The query contains three parts: the two subqueries and the higher query that combines these two. What each of those subqueries are doing is selecting some node in the graph of which $R$ gives the edges. It then counts how many outgoing edges each of these nodes have. It then compares these counts in the higher query. We can consider this to be the following query:

$$\text{COUNT-COMPARISON} = \{(C_1, C_2) \mid (C_1, C_2) \in T \text{ and } \varphi_{CC}(C_1, C_2)\} \quad (5.2)$$



(a) $R^{\boldsymbol{A}}$ describes two completely connected components of the same size $n$

(b) $R^{\boldsymbol{B}}$ describes two completely connected components: one of size $n$ and one of size $n+1$

Figure 5.1: Two graphs represented by R that will be compared in an EF-game

where we assume that $\varphi_{CC}(C_1, C_2)$ is a formula that is true if $C_1$ has fewer outgoing edges in $R$ than $C_2$. Since this can easily be converted to a first-order formula, we shall prove with an EF-game that $\varphi_{CC}(C_1, C_2)$ cannot be expresses in first-order logic.

The situation can be simplified if we assume that the connected components of both $C_1$ and $C_2$, are both completely connected, at which point it becomes the same as counting the amount of nodes in the connected component and comparing those. In Figure 5.1 two possible cases of $R$ are shown. The first structure is $\boldsymbol{A}$, shown in Figure 5.1a, where $R$ represents two fully connected components of the same size $n$. The second structure, called $\boldsymbol{B}$, is shown in Figure 5.1b. Here $R$ represents two fully connected components but one is of size $n$ and the other of size $n+1$. $\varphi_{CC}(C_1, C_2)$ should not be true for any single pair of nodes in $\boldsymbol{A}$. In $\boldsymbol{B}$ it should be true for all pairs of nodes where $C_1$ is a node from the component of size $n$ and $C_2$ a node from the component of size $n+1$. The proof will now proceed to prove that with an $n$-move EF-game it is impossible to differentiate these two structures and that the Duplicator thus has a winning strategy. To make this proof easier to read, we shall call the connected components of $\boldsymbol{A}$ $a_1$ and $a_2$. The smaller component of $\boldsymbol{B}$ will be called $b_1$ and the larger component $b_2$.

This proof shall proceed by describing the winning strategy of the Duplicator. When the Spoiler picks its first element from one of the two connected components of one of the two structures, pick a connected component in the other structure. Assume that nodes picked were from $a_1$ and $b_1$, then if in any proceeding move the Spoiler picks an element from $a_1$ or $b_1$ the Duplicator responds with an element of $b_1$ or $a_1$ respectively. After $n$ moves the Duplicator can at most have picked out all the nodes from one connected component without knowing if there are more nodes in the other connected component. Thus, the outputs of some FO formula of quantifier rank $n$ must output the same

value in both structures. This means that either for both $\boldsymbol{A}$ and $\boldsymbol{B}$ it must output all pairs of nodes with $C_1$ from $a_1$ or $b_1$ component and $C_2$ from the $a_2$ or $b_2$ component or nothing at all. Either choice will be wrong in one of the two structures. Since for any quantifier rank such a pair of structures can be devised, it is impossible that such a formula actually exists. The exact same reasoning works for the initial pairing of $a_1$ and $b_2$, $a_2$ and $b_1$ and for $a_2$ and $b_2$.

The structures used in the preceding proof also show that this is clearly related to a majority query, which returns which of two relations is larger. This can be seen if we were to consider each of the connected components to be pairs of nodes stored in separate relations. The majority query is also known to not be FO-definable, thus it makes sense that our query comparing sizes can also not be expressed in FO logic.

### 5.2.2 Relational Tree-Algebra

We have now demonstrated that FO logic, and thus relational algebra, is not sufficient to express the count comparison query. A reasonable question then would be to ask how the relational algebra could potentially be expanded to make this query definable in that logic. A first logical expansion would be to look at what the relational meta-algebra did on the relational algebra, and repeat this with trees. We choose to introduce tree operators since that is the complex data structure we are using in the count comparison query. This algebra could also potentially allow us to later express the similarity relation in it, giving us another good reason to investigate it. Many of the operators are also similar to operators from the meta-algebra, be they primitive or derived operators.

#### Different possible extensions

The first new operator we will define, is the '$match_{a,i}$' operator, which will filter all rows for which the trees in column $i$ have the root label $a$. This is similar to the match operator in relational meta-algebra, which is a derived operator that can be defined using the extract a select on that result and then projecting the extracted column away [NVVV99]. The next operator is the '$extract_{i,j}$' operator, which will extract the $j$-th child and its entire subtree from the tree in the $i$-th column and add it as a new column after all currently existing columns. Again this is similar to the '$extract_{i:j}$' operator from the relational meta-algebra, which extracts the all subformulas of arity $j$ from the formulas in column $i$. The final primitive operator we will define is the '$construct_{a,i_1,\ldots,i_k}$' operator which will create a tree with a new root node labeled '$a$' with $k$ children with for the $j$-th child, the tree found in column $i_j$ and then add it as a new column. This is again similar to the derived '$contruct_{\alpha}$' operator which creates a formula $\alpha$, replacing the references to columns in it with the actual formula in that column and adding that as a new column at the end. In the relational meta-algebra this operator is derived since it can be expressed using an extract operator, a wrap

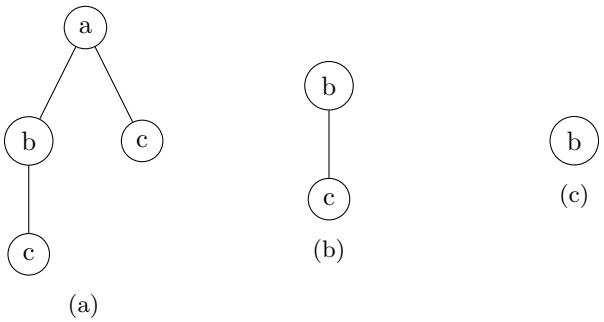



Figure 5.2: Three trees to be used as an example of the operators of the relational tree-algebra

operator and rewrite operators [NVVV99]. Finally, we also have to extend the classical equality operator, allowed in formulas of the selection operator, so that it will also check whether two trees are identical. We shall call this extension to the relational algebra with these operators and the extension of the equality operator the *relational tree-algebra*. When comparing this to the other types of relational tree-algebras, we sometimes refer to is at *general relational tree-algebra*. Sometimes we shall shorten it as *RTA* in the general case or *GRTA* when comparing it to other types of RTA's.

To make it easier to understand these operators, consider the following examples. Firstly for the 'match' operator consider that all the trees in Figure 5.2 and assume that they are in the second column of a relation $S$. If we then do $match_{b,2}(S)$ we would only keep the row containing the tree in Figure 5.2b, since this is the only one with a b label in the root node. If we used $extract_{1,2}(S)$ for the row with the tree in Figure 5.2a it would add the tree in Figure 5.2b as the value for the new column. Finally, assume that the trees in Figure 5.2b and 5.2c are in columns 2 and 3 of the relation $V$. The application of $construct_{a,2,3}(V)$ results in the tree in Figure 5.2a.

A valid question that could now be asked is whether we aren't making this algebra too strong by not enforcing any kind of typing. The issue is that there isn't a logical general typing that could be enforced, unlike in the relational meta-algebra where the arity of inputs of the operators determine the arity of the outputs. When working with trees however, there is not such a concept immediately available. It would be possible to argue that the amount of children would define a similar concept to arity, since it would determine how high $j$ in $\text{extract}_{i,j}$ would be allowed to go. Like arity in the relational algebra, in the RTA there are only a few operations that can create a new tree that has a different amount of children: extract and construct. For construct the amount of children in the new tree is dependent on the amount of columns passed to it as parameters. For extract the amount of children is dependent on the structure

itself, since it depends on what child you extract and how many children it has. This also shows the difference between arity and the amount of children: with arity we can deduce the outcome of every operator simply by knowing its arity of its inputs, with the amount of children this is not the case. However, this could be solved by defining a new, more strict algebra to more closely match the relational meta-algebra.

Our new and more restrictive algebra would have the same operators as the previous algebra but with slight modifications to each one to make sure they always output the same type. The '$match_{a,i}$' operator remains unchanged since it does not modify the amount of children. The '$\text{extract}_{i,j}$' operator is changed to the '$extract_{k,i,j}$' where $k$ is a positive integer number that represents the amount of children the child being extracted must have to be returned. Finally, '$construct_{a,i_1,\ldots,i_k}$' also remains unchanged since the amount of children a tree will have directly depends on the amount of parameters given to it, which in this case is $k$. On top of these changed operators we would now have to assume that each column in a relation has a type. This would be written similarly to the types in the relational meta-algebra. However, in this case the numbers between the $\langle\rangle$ represent the amount of children the root node of a tree in that column has. We call this restriction *child count relational tree-algebra* or *count relational tree-algebra*. We shall sometimes shorten this to *CCRTA*.

Now that this first restriction has been defined, it could be considered whether this restriction is strong enough; typically the stronger the restriction the less complex a problem within it becomes. An alternative that could be proposed, is an algebra that is based on some *context-free grammar*, often shortened as *CFG*. In this case it would be assumed that the leaves of the tree are the terminals symbols and the non-terminal symbols are the internal nodes. The type we would then use for tree values is either a production rule or a terminal symbol. Of course to make this more flexible, a non-terminal could be allowed to be some commonly known type, like a string or an integer. This would allow us to store actual data in these trees without having an extreme amount of terminal symbols. If the type of a column is a production rule, then the trees in that column can only have root nodes that are created with that production rule. If the type of a column is a non-terminal, then it is a column in which each value represents a single node of that type. For the operators, firstly '$\text{match}_{a,i}$' has become obsolete since the name of the root node is always known, since it is defined by the rule used to produce it and that is the type of the column in which a tree is stored. The operator '$\text{extract}_{i,j}$' becomes the operator '$extract_{p,i,j}$' where the $j$-th child is only extracted if it is produced by the rule $p$. Finally, the operator '$\text{construct}_{a,i_1,\ldots,i_k}$' is modified to the operator '$construct_{p,i_1,\ldots,i_k}$' where $p$ is the rule that produces the new root and for each $i_j$ with $1 \leq j \leq k$ the $i_j$-th column has a type of which either the right-hand side of the rule is the same non-terminal symbol as the symbol in the $j$-th spot in rule $p$, or it is the same terminal symbol as the symbol in the $j$-th spot in rule $p$. We call this

kind of restriction to the relational tree-algebra the *CFG relational tree-algebra*, which we sometimes shorten to *CFGRTA*.

Note that the CFGRTA only works if a CFG is given to which all trees adhere. This will not always be the case. However, it does not sound unreasonable to assume that the trees all have some sort of known structure and thus that there would be some semantics for these trees. The trees could be defined to be reasonable and interpretable if they adhere to some syntax, which could be given by a CFG. Thus assuming that the contained trees have some predefined semantics that will always hold true, it is not unreasonable to assume that some context-free grammar exists for these trees and is defined.

### Equivalence of extensions

Now that we have defined these algebras, it is an interesting exercise to relate them to each other to compare which is more expressive and which are equally expressive. The main difference between the CFG relational tree-algebra and the other extensions, is that it assumes there is some CFG to which all trees adhere. This assumption is not made by the other two extensions. This of course makes it less expressive, since it cannot express queries over trees of an arbitrary shape. However, should we assume that all trees adhere to some context-free grammar, then all these extensions offer the same expressive power, as long as we assume that each relation in each of the algebras behaves according to the type system of that algebra. Of course if for example we would apply the algebra based on CFG's with the relations whose types are the amount of children, then clearly a match operator could not be created, since it was removed in the algebra based on CFG's due to its redundancy with types. Also note that each more restrictive type is also a valid type of the less restrictive type before it. As an example a column that is restricted to only allow root nodes produced by a specific rule of a CFG, will always have the same amount of children, which would thus be its type in the count relational tree-algebra. For the CCRTA this is trivial, since the only type of column that the general relational tree-algebra knows is a column that is or is not a tree. This thus means that if we have a relation which is valid in the case of the extended algebra based on CFG's, it is also a relation valid in the other extensions of the algebra. Thus, we shall prove their equivalence over relations that would be correctly typed for the CFG relational tree-algebra.

To prove the equivalence of the different algebras, we will describe their operators in each other. First we will describe how the CFG and the children count relational tree-algebra are equivalent. After this we will prove the same for the general and the children count relational tree-algebra. These two equivalences combined then also prove that the general and CFG relational tree-algebra are also equivalent.

**CFGRTA and CCRTA** The first operator we shall discuss is the 'match' operator. In the CFG relational tree-algebra this operator is redundant due to the typing of relations. As a result it would not be necessary to define this operator in the other languages, since this could simply be determined by checking the column that is being used. However, to make it easier to define future operators, we shall define a match operator that filters all rows that are produced by a certain rule of a CFG. In the children count relational tree-algebra, we would define this kind of filter by doing an extraction for each child in the rule followed by a match on the label of child in this rule that is being extracted. Finally, we would project the added columns away again. However, the issue here is that a non-terminal symbol in a CFG may have multiple rules that can produce it, which can have a varying amount of children. To solve this we would take a union of all the different combinations of the sets of possible children counts that a particular symbol in the given CFG has.

To understand this better, assume that the rule we are attempting to match is 'A $\rightarrow$ B C' where B can either have 2 or 4 symbols on the right-hand side and C can have 2 or 3 symbols on its right-hand side. This would mean that each row of $\{2,4\} \times \{2,3\}$ would represent a possible combination of the amount of children for each of the child nodes with labels B and C. This means that if the root node has the label A, the first child has the label B and has 2 children and the second child has the label C with 2 children it could be produced by A $\rightarrow$ B C. This also holds if we fill the child counts in with any of the combinations found in the rows of $\{2,4\} \times \{2,3\}$.

More generally, assume a rule with $k$ symbols on its right-hand side, where the $j$-th symbol has the set $C_j$ of possible amount of children. The possible combinations of amount of children are then the rows of $C_1 \times \cdots \times C_k$. After checking all these potentials we must filter out those trees which have the correct amount of children and not at least the required amount of children for the root node. To do this we construct a tree with a temporary root node and extract only its only children with $k$ children themselves. Finally, the match operator would be written as

$$\begin{aligned}\text{match}_{p,i}(R) \equiv &\pi_{1,\ldots,n}(\text{extract}_{k,n+k+1,1}(\text{contruct}_{temp,i} \\ &\Big(\bigcup_{j_1,\ldots,j_k \in C_1 \times \cdots \times C_k} (\text{match}_{s_k,n+k}(\text{extract}_{j_k,i,k}(\ldots \\ &(\text{match}_{s_1,n+1}(\text{extract}_{j_1,i,1}(\text{match}_{s_0,i}(R)))))))))\Big)\end{aligned} \tag{5.3}$$

with R either a relation or some formula in the child-count relational tree-algebra with arity $n$, $p$ a CFG rule with symbols $s_1, \ldots, s_k$ as its right-hand side, $s_0$ as the symbol on its left-hand side and $i$ the column being matched. To describe the match operator of the CCRTA in the CFGRTA all we do is simply check the column in which the query is returned. If this is a column with the same symbol in the left-hand side of its rule as the label in the match clause, then nothing is

changed. If the symbol is not the same as the one in the 'match' operator, then $\times\emptyset$ is added to remove all rows since not a single row would match.

Next we will describe the 'extract' operator. To describe the '$\text{extract}_{k,i,j}$' operator of the CCRTA in the CFGRTA, we start by looking at the type of the column of $i$. Next we take the $j$-th symbol on the right-hand side of that rule. If the rule does not have at least $j$ symbols in its right-hand side, then we simply add $\times\emptyset$ since nothing could be returned. If there is a $j$-th symbol, then we take all rules which have this symbol as their left-hand side and that have $k$ symbols in their right-hand side and call this set of rules $P$. Thus, we can write

$$\text{extract}_{k,i,j}(R) \equiv \bigcup_{p \in P} \text{extract}_{p,i,j}(R) \tag{5.4}$$

for some relation of formula $R$ and $P$ as described above. Note that this is only a formula in CFGRTA if the set $P$ is actually finite. This is guaranteed if the CFG is finite but not if the CFG is infinite.

To define the '$\text{extract}_{p,i,j}$' operator of the CFGRTA in the CCRTA we will extract all $j$-th children with $k$ children themselves, where $k$ is the amount of symbols in the right-hand side of $p$. Then we will use the match, defined in the previous paragraph, to match on rule $p$ This thus results in the formula

$$\text{extract}_{p,i,j}(R) \equiv (\text{match}_{p,n+1}(\text{extract}_{k,i,j}(R))) \tag{5.5}$$

where $R$ is some formula or relation or arity $n$, $p$ is the rule and $k$ the amount of symbols on the right-hand side of $p$.

The final operator left to prove is the 'construct' operator. This operator becomes rather trivial to prove, since we assume that the relations use a CFG as their type. This means that the operation is heavily restricted in the count relational tree-algebra, since it may only construct trees that adhere to the CFG. Thus, in the CCRTA, assuming the CFG rule $p$ has $s_0$ as the symbol on its left-hand side and the symbols $s_1, \dots s_k$ as its right-hand side, the formula would be

$$\text{construct}_{p,i_1,\dots,i_k}(R) \equiv \text{construct}_{s_0,i_1,\dots,i_k}(\text{match}_{s_1,i_1}(\dots(\text{match}_{s_k,i_k}(R)))) \tag{5.6}$$

for $R$ some formula or a relation.

In the other direction, writing the 'construct' operator of the CCRTA in the CFGRTA, we simply use the rule that is used to produce the new tree. This is possible because all trees must still adhere to the CFG. This could easily be determined by looking at which columns are being used, looking at the left-hand symbols of their types and then choosing the single rule that could produce a tree with the new given root node. Should such a rule not exists, then the tree would no longer adhere to the CFG and thus could not have been created in the first place.

**CCRTA and GRTA** Now the same exercise as in the preceding paragraphs shall be done between the general relational tree-algebra and the count relational tree-algebra. Firstly the '$\text{match}_{a,i}$' operator which is trivial since it is not changed between the two extensions. Next we shall do the same for the '$\text{construct}_{a,i_1,\ldots,i_k}$' operator which is equally trivial as it is also the same between the two extensions.

Finally we shall show the equivalence of the two extensions regarding the 'extract' operator. To express the extract of the GRTA in terms of the CCRTA, we look at the CFG and determine all the different lengths of the right-hand sides of all rules, call this set of lengths $C$. The set extract operator can thus be written as follows:

$$\text{extract}_i(r) \equiv \bigcup_{c \in C} \big(\text{extract}_{c,i}(R)\big) \tag{5.7}$$

for some formula or relation $R$ and with $C$ as described above. Note, similar to the 'extract' operator of the CFGRTA in the CCRTA, this is not a GRTA formula if $C$ is infinite. $C$ is again guaranteed to be finite if the CFG is finite but can be infinite if the CFG is infinite.

**Equivalences** The previous two paragraphs prove that the general relational tree-algebra and the count relational tree-algebra are equivalent if there is a given CFG to which all trees adhere. The paragraphs before that showed that the count relational tree-algebra and the CFG relational tree-algebra are equivalent if all trees adhere to a CFG and each column only contains trees that can be produced by a single CFG rule. On top of this, both proofs made assumptions that the CFG to which all trees adhere. The first of these assumptions was that there is not an infinite amount of rules with a certain amount of symbols on the right-hand side. The second of these assumptions was that the amount of symbols in each of the right-hand sides the rules is finite. If we now apply the restriction that each column only contains trees that can be produced by a single CFG rule, to the equivalence between the general and the count relational tree-algebra, the equivalence still holds. Thus, it is also possible to show the equivalence between the general relational tree-algebra and the CFG relational tree-algebra in this situation, by translating it to the CCRTA as an intermediate language first and then translating it to the other extension.

Finally, now that the equivalence has been proven, it is also clear to see that if there was no CFG present many of these equivalences could no longer be done. For example the CFG relational tree-algebra would no longer be equivalent to the count relational tree-algebra. This is because the latter could simply create something that does not adhere to the CFG, which is impossible to express in the CFG relational tree-algebra. Similarly, the extract of the general relational tree-algebra would no longer be expressible in the count relational tree-algebra. This is because there is no limit to how many children that extracted child may

have. Note however that the operators of the more restrictive extensions can all be expressed in the general relational tree-algebra. Without the restrictions however, the more restrictive extensions cannot express the GRTA operators. This means that the more restrictive operator can be used in the general relational tree-algebra, but they are simply not primitive operators.

Now that we have described the various algebras we should also consider how difficult it would be to adapt our query to them. While it would not be a stretch to assume that there would be some CFG to which all our trees of code would adhere, the issue with both the CCRTA and the CFGRTA is their typing. Assume we used the amount of children underneath the root of the tree as the type for our columns. Then for each pair $i, j$ of the possible amount of children we need to create a new relation that represents that a tree with $i$ children is similar to a tree with $j$ children. This would still be limited by the CFG, but it would be cumbersome nonetheless. For the typing that uses the rules of a CFG this problem is even worse. This thus means that due to its loose typing restrictions the GRTA would be the most natural to choose.

**An equivalent calculus**

So far we have described three different possible extensions, that under certain circumstances are equivalent, but each with their normal restrictions they are not equivalent. Now we will define an equivalent calculus for the general relational tree-algebra. This is because it seems the most appropriate for our use-case and a calculus make it easier to prove that something cannot be expressed. However, note that the operators of the more restrictive extensions are also expressible in the GRTA as derived operators. These operators can be achieved by following a similar translation to the one done for the algebra extension. This means that if something cannot be expressed in this calculus, it will also not be possible to express it in the more restrictive calculus and thus also not in their algebra. Thus, we shall create an equivalent calculus defining only the primitive operators of the general relational tree-algebra.

We shall start from the relational calculus, since that already covers the translation of the entire relational algebra on which this extension was based. The first thing to add is to add the concept of tree variables and tree types, where tree variables are variable that can only contain values that are of a tree type and a tree type simply means that the value is a tree. As a result of this new type we shall also expand the use of the existential and the universal quantifiers to allow for quantification of trees. This addition makes sense because an existential quantifier is how a projection is emulated in relational calculus. On top of this because we can project tree columns away, we need to be able to quantify tree variables. We also need to extend the equality relation '=' so that it can also compare tree variables and constants that represent trees. Two trees will be equivalent if their structure is the same and the labels on the pairs of nodes in the same location in the structure are the same. Next we define the

operators starting with 'match'. This operator we shall define by introducing a set of new predicates '*match-a*$(x)$' where $a$ is the label that the root node must have and $x$ is some tree variable or a constant that represents a tree. The next operator is the 'extract' operator, which will be translated to the predicates '*extract-j*$(y, x)$' with $j$ a natural number indicating which child to extract, $x$ some tree variable or a constant that represents the tree to extract from and $y$ some tree variable or a constant that represents the extracted tree. Finally, we need to convert the 'constrict' operator, which we shall do by introducing the predicates '*construct-a-k*$(y, x_1, \ldots, x_k)$' where $a$ is the label to assign to the new root, $k$ is the amount of children this node will have and for each $1 \leq i \leq k$ $x_i$ is either a tree variable or a constant that represents a tree and $y$ is the tree with root node $a$ and with as $i$-th child $x_i$. We shall call this newly defined calculus, the *relational tree-calculus*. We shall sometimes shorten it to *RTC*.

In the previous paragraph we described a calculus for the general relational tree-algebra. We can also define a relational tree calculus for the CFGRTA and the CCRTA. If we are comparing these new calculi, then the previously defined RTC becomes the *general relational tree-calculus*. This shall then be shortened to *GRTC*. For the CCRTA we will define a set of predicates for each of the previously defined predicates. As an example, for the match operator we will define '*match-a-m*$(x)$' which is only true if the root node of the tree of $x$ has $m$ children and the label $a$. We will define a similar set of predicates for the 'extract' predicates. The 'construct' predicates do not need to be updated. This new calculus will be called the *child count relational tree-calculus* or *count relational tree-calculus*. This will be shortened to *CCRTC*. A similar set of predicates can also be defined for the operators of the CFGRTA. Instead of using the child count $m$, these then use predicate $p$ which must produce the root of the tree of $x$. Additionally, since the CFGRTA does not allow any values that cannot be produced by the CFG, we will not allow trees that cannot be produced by the CFG to be quantified. This then results in the *CFG relational calculus*, sometimes shortened to *CFGRTC*.

### Definability of query

Now that a calculus has been established it is possible to prove that the relational tree-algebra is not powerful enough to express the count comparison query. This proof is similar to the one done in Section 5.2.1, but the definability must be checked again since new operators have been added. These new operators could have potentially added enough expressiveness to make our query suddenly expressible. We can use an EF-game to prove that this calculus is still not strong enough to decide $\varphi_{CC}(C_1, C_2)$. For this proof we shall reduce a query comparing the size of two structures to $\varphi_{CC}$. We shall then show that given $n$ moves for the EF-game that there will always be two structures that cannot be differentiated by any formula for that query. This thus shows that it is impossible to create a formula for this query in our calculus and thus by extension in our relational tree-algebra. Since we could solve this with $\varphi_{CC}$, this also means that $\varphi_{CC}$ can

also not be expressed in the relational tree-algebra. To set this up, we will use a simple trick to allow us to work with trees as if they were natural numbers: all trees in our instances will be chains with a single label. The length of these chains denotes the natural number they represent.

The vocabulary over which we will define our game will be $\sigma = (U_1, U_2)$ with both $U_1$ and $U_2$ a relation of arity one and its only column a tree type. The query over this vocabulary we shall consider is $\varphi_{U_1\text{-}smaller}(u_1, u_2)$. This query will contain all pairs of elements from $U_1$ in $u_1$ and $U_2$ in $u_2$ if the chain of $U_1$ is smaller than that of $U_2$. We shall define two $\sigma$-structures for our proof and call them $\boldsymbol{A}$ and $\boldsymbol{B}$. These structures will be defined to be used in the $n$-move EF-game. $U_1^{\boldsymbol{A}}$ will contain $2^n + 1$ trees and $U_2^{\boldsymbol{A}}$ will contain $2^n$ trees. Both $U_1^{\boldsymbol{B}}$ and $U_2^{\boldsymbol{B}}$ will contain $2^n$ trees. Each of these trees in $U_1$ builds upon the previous one by simply constructing a tree with a root node with the label $a$ and giving it the previous tree as its only child. Thus, all trees are simply chains with only a single label $a$. For the trees in $U_2$ the same is done but with the label $b$. This also means that two consecutive trees can be linked to each other by a 'contruct-$a$-1$(x, y)$' predicate if they are in the $U_1$ relation. The same also goes for the trees of $U_2$ but with the 'contruct-$b$-1$(x, y)$' predicate. Any two consecutive trees can also be linked together with an '$extrac$-1$(x, y)$' operator with the $x$ containing the chain of one length shorter of the same relation. An important thing to note is that due to the different labels in the trees of $U_1$ and $U_2$ it is not possible to relate trees between the two relations with our current operators.

Next we shall give the winning strategy for the Duplicator. The Spoiler starts by picking the $i$-th tree from $U_1$ from one of the two structures. If $i$ is in the first half of the structure, the Duplicator chooses the $i$-th node in the other structure. If $i$ is in the second half of the structure, choose the tree in the other structure that is the same amount of trees away from the last tree. The reason the Spoiler only picks from $U_1$ is that if it picked from $U_2$ the duplicator could always answer with the element in the same position in the other $U_2$. On top of this, the Duplicator cannot choose an element from $U_2$ after the spoiler picked one from $U_1$, as this would break the partial isomorphism, making the Duplicator lose the EF-game.

Next the Spoiler will pick another node from $U_1$. Assume this node is more than $2^{n-2} - 1$ trees away. Then we can simply take the node in the other structure that is the same amount of trees away from the end tree in the same direction. If the node is less than or equal to $2^{n-2} - 1$ trees away, then pick the tree at the same distance in the same direction in the other structure. This logic holds for any pick by the Spoiler from $U_1$ in round $i$ of the game. Assume the closest previously chosen, the first and the last node in $U_1$ are all more than $2^{n-i} - 1$ trees away. Then simply pick a tree that is $2^{n-i}$ trees away from the closest tree and the start and end node in the other structure. The reason for

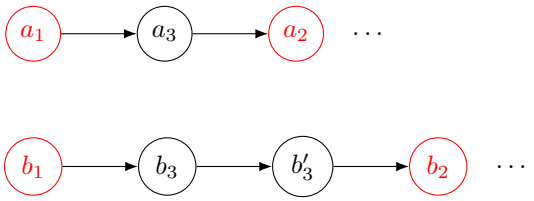



Figure 5.3

this skipping over $2^{n-i}-1$ nodes is that $2^{n-i}-1$ is the size at which an $n-i-1$-move EF game could differentiate between the relations $U_1$ if the nodes at either end are already chosen. It would do this by constantly skipping over $2^{n-i}-1$ nodes for the remaining moves inside this section with already chosen start and end nodes. If the Duplicator follows this move with a same size gap, the Spoiler continues skipping from the start towards the end. If the Duplicator does not leave the same size gap, the Spoiler will continue this strategy. However, now it plays between the start and end nodes defined by the smaller of the two gaps created the previous turn.

Notice that the spoiler can step $2^{n-i}-1$ nodes on the $i$-th turn of the $n$ move EF-game. However, now it only has $n-i$ moves remaining, which means it can only leave gaps that it could win in $n-i-1$ moves. The distance crossed by all of these remaining $n-i-1$ moves combined, is of course exactly the size of the gap left at the $i$-th move. Thus, in an $n$-move EF game, the Spoiler can move a total of $2^{n-2}+2^{n-2}-1 = 2^{n-1}-1$ from its starting node and force the Duplicator to move in the exact same size and direction with it. This means that the Spoiler can win as long as both $U_1$s are smaller than $(2^{n-1}-1)*2+1 = 2^n-1$. This is because the Duplicator can choose its first node in the exact middle and then move either direction.

To make it more clear how exactly this proof works, consider the graph in Figure 5.3. This graph represents a section of $U_1^{\boldsymbol{A}}$ and of $U_1^{\boldsymbol{B}}$, where the start and end nodes have already been chosen by the spoiler and Duplicator. Again the edges in this graph show that the tree at the source of an edge is used to construct the tree at the destination of the edge. The first pair of nodes chosen are $a_1$ and $b_1$ from $U_1^{\boldsymbol{A}}$ and $U_1^{\boldsymbol{B}}$ respectively. In the next move $a_2$ is chosen by the Spoiler and $b_2$ is chosen by the Duplicator as a result. We will now show that the Duplicator should not have chosen $b_2$, since the spoiler can now win in a 1-move EF game on the section between the pairs of red nodes. The Spoiler can obviously win by simply choosing $a_3$ and then the Duplicator cannot reply since both $b_3$ and $b'_3$ would only connect it to one of the previously chosen nodes but not both.

Another example to show how this recursion works is Figure 5.4. Here $a_1$, $b_1$, $a_2$ and $b_2$ have all been chosen like how they were in the last time. Now we

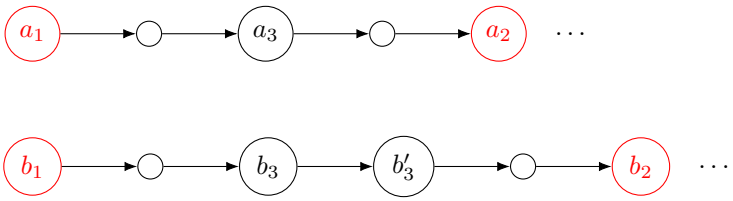



Figure 5.4

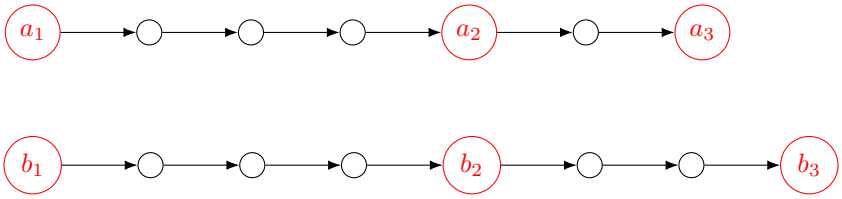



Figure 5.5

will demonstrate that again choosing $b_2$ was a mistake by the Duplicator, since the Spoiler can now win the 2-move EF-game. To do this the Spoiler chooses $a_3$ and the Duplicator has to respond with either $b_3$ or $b'_3$. Consider the state of our game after this round, assuming $b_3$ was picked. If we now consider $b_3$ and $a_3$ to be our start nodes we see that the situation is the same as in Figure 5.3. This shows how the recursion of our approach in the proof works. If $b'_3$ was chosen we make it and $a_3$ our end nodes and then make the same argument.

As a final example, first take the graph in Figure 5.5. This shows how the Spoiler would win in a 4-move EF game. First the Spoiler takes picks a node $a_1$, which the Duplicator matches with $b_1$. Next the Spoiler picks $a_2$ which the Duplicator has to match with $b_2$ or lose, as we have shown in the previous paragraph. Then for its third move, all that remains is essentially the same as after picking only $a_1$ and $b_1$ in Figure 5.3. Thus, again here we have already proven that the Spoiler has a winning strategy. This means that the Spoiler for the 4-move EF-game can move 7 nodes from its first node picked and still win. Note that this move of 7 nodes only works if $b_3$ is in fact the last node. If it is not then the Spoiler can only move 6 nodes and the Duplicator would win.

Finally, take the graph in Figure 5.6. This is another example of a graph where the Spoiler can win the 3-move EF-game with $a_1, b_1, a_2$ and $b_2$ already chosen. Again here we'd play essentially the game of Figure 5.5 between the new start and end nodes. The Spoiler starts by choosing $a_3$ which we proved earlier means the Duplicator needs to play $b_3$. These are the same as $a_2$ and $b_2$ in Figure 5.5 between the new start and end nodes. Next the Spoiler plays $a_4$ which the Duplicator must follow up with $b_4$. This is similar to $a_3$ and taking the node left of $b_3$ in Figure 5.5. As the final move the Spoiler chooses $a_5$, to

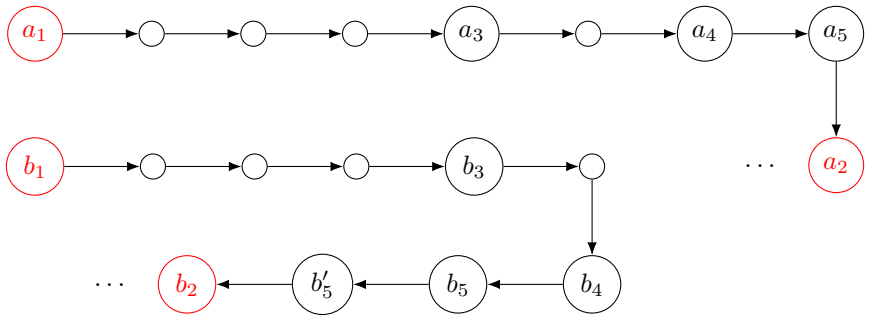



Figure 5.6

which the Duplicator cannot reply. This is because $b_5$ and $b'_5$ both lack one edge to previously chosen nodes compared to $a_5$. This shows that the Duplicator can pick a node 8 nodes further, skipping 7 nodes. Thus, when there are $i$ moves remaining and the Spoiler in the $i$-move EF-game can move a distance of $d$ from its start node, it can choose the next node $d+1$ nodes further, skipping $d$ nodes.

Finally, after the first two moves have been played, the Spoiler can move an additional distance, which is the same it could in the 4-move EF-game. This is because we now have 3 rounds left, which would be the same as having the first node already be chosen in the 4-move EF-game. In general if in the $n$-move EF-game two rounds have been played, the Spoiler can still travel the same distance as the Spoiler could travel from its start node in the $n-1$-move EF-game. Assume this distance is $d$. This means that in an $n$-move EF-game the Spoiler can move $(d+1)+d = 2d+1$ nodes from its start node. In our example that means that in the 4-move EF-game, the Spoiler can move a distance of $2*7+1 = 15$ nodes.

This proof thus shows that for any formula using $n$ quantifiers, a pair of structures can be devised for which the Duplicator wins the $n$-move EF-game. This means that the formula $\varphi_{U_1\text{-}smaller}(u_1, u_2)$ should either in both $\boldsymbol{A}$ and $\boldsymbol{B}$ return all pairs of nodes from $U_1$ and $U_2$ or nothing at all. However, this causes a contradiction, since we know that each of these outcomes is wrong in either $\boldsymbol{A}$ or $\boldsymbol{B}$. Thus, such a formula with at most $n$ quantifiers does not exist. Since we have given a construction for every $n$, we have also proven that such a formula does not exist. Now assume that $\varphi_{CC}(C_1, C_2)$ could be expressed. This would mean that $\varphi_{U_1\text{-}smaller}(u_1, u_2)$ could also be expressed. This is because we could replace each reference in $\varphi_{CC}(C_1, C_2)$ to $R$ with $(U_1(C_1) \wedge U_1(C_2)) \vee (U_2(C_1) \wedge U_2(C_2))$. Then if $\varphi_{CC}(C_1, C_2)$ returns a node with the label $a$ on the root node of $C_1$ and $b$ in the root node of label $C_2$ we return this pair as a result of $\varphi_{U_1\text{-}smaller}(u_1, u_2)$ with $C_1$ being the value of $u_1$ and $C_2$ the value of $u_2$. This is a correct because, as we showed in Section 5.2.1, counting the outing edges of a completely connected component is the same as counting the amount of nodes in it. This means that

$\varphi_{CC}(C_1, C_2)$ cannot be expressed because it would allow $\varphi_{U_1\text{-}smaller}(u_1, u_2)$ to be expressed, which we know is impossible.

Note that this proof also still holds for the other more restrictive extensions, since the CFG would only contain two rules and all trees root nodes are produced with only one of those rules, the other one is for termination. For the CCRTC it is also straightforward: there is only ever one child count in all the trees, thus there is only one set of operators useful, which are those operators dealing with trees with a single child. Thus, in both cases the operators can be mapped to the more board ones of the general relational tree-calculus and the proof would go the same.

Another note to make is that the proof so far has assumed that the trees created by the existential quantifier, are all trees in our relations. However, this need not be the case. It is possible that an existential quantifier quantifies an entirely new tree that is not part of any relations in the structure. This also means it would be possible to in one move pick every tree in one move by picking a tree with some new root node and then the $i$-th child is the $i$-th element of $U_1$. Then in one move the Duplicator would have to also create a tree that responds with a tree that does the same but for the trees from $U_1$ of the other structure. This would allow for the Spoiler to immediately pick all elements of $U_1^{\boldsymbol{A}}$ to which the Duplicator would have to respond with the same tree for $U_1^{\boldsymbol{B}}$ with an extra tree to make sure to preserve the relation created by the 'construct' operator. This extra tree could then be exploited by the Spoiler by extracting the last child in the large tree in $\boldsymbol{A}$. The Duplicator would also have to respond with the last child in the other tree, thus breaking the partial isomorphism and losing the EF-game. If the Duplicator did not, then the Spoiler would simply choose the child after the one the Duplicator chose in the tree. The Duplicator would have to follow with a child that follows the last child that the Spoiler chose, which is impossible.

However, this problem with the quantifier is not an issue. The amount of moves of an EF-game depends on the quantifier depth, but the size of the structure can be determined by the formula. Normally this is simply done by looking at the quantifier depth of the formula using that to determine the size of the structure. However here we will look at the formula and determine the largest child extracted. Call this $c$. We would then simply consider each move of an $n$-move EF-game to be $c$ moves in our current structure. This is because the Spoiler could only extract the first $c$ children of any tree at any depth. This means that the Duplicator is at max only forced to duplicate $c$ trees in one move. Thus, to have a structure where the Duplicator still wins, we would create a structure that would win on $c * n$ moves. This could easily be done by creating structures where both $U_1$s and $U_2$s are larger than $2^{n*c} - 1$. We could even lower this size requirement. Rather than considering the highest of children being extracted, we consider all unique indexes of children being extracted. Call

this value $c_u$. It is obvious that $c_u \leq c$ and thus a lower upper bound would be if all relations are at least $2^{n*c_u} - 1$ in size.

Finally, note that this is not a problem with the CFGRTC, since every tree must adhere to our CFG, even the ones created by the existential quantifiers. Thus, the above could never arise in the CFG relational tree-calculus. The child count relational tree-calculus would use the exact same solution as the general relational tree-algebra uses. This is because at any point it knows how many children any root node has and inspection into the formula will result in the same as in the previous paragraph.

### Adding more operators

Now that we have shown that our initial extension to the relational algebra did not suffice to add the ability to count and see which of the relations is larger, it makes sense to try and look for extensions that would make this possible. A logical first extension to look at is *subtree containment*. Subtree containment is the ability to check if something is a subtree of something else, as defined in Section 3.1.3. For our extension a subtree containment operator would filter out the rows for which a tree is a subtree of some other tree. A subtree operator would make sense in our situation, given that it could be used to define similarity between two trees. This operator would then be added in the shape of '$\text{subtree}_{i,j}$' where all rows are returned in which column $i$ contains a tree that is a subtree of the tree in column $j$. This essentially provides us the transitive closure of the same relation we could get with the 'construct' operator in our previous proof that a count comparison cannot be expressed in our extension. We call this new extension on the relational tree-algebra the *relational subtree-algebra*. We shall sometimes shorten this to *RSA*.

Note that the way this operator is defined, it will work as an additional operator in any of our previously defined extensions. However, similar to the 'match' operator, it may be redundant for some cases in the CFG relational subtree-algebra. This is because at any time we know the rule of the CFG that produces all trees in a particular column. Thus, if we are checking that trees produced by a rule $p_i$ are subtrees of trees produced by rule $p_j$, we may already know this from the CFG. It is indeed sometimes possible to determine that a tree produced by $p_i$ could never appear under a tree produced by $p_j$. As an example say we have a CFG consisting of four rules: $a \to b$, $b \to b$, $b \to c$ and $b \to c$ with $c$ and $d$ terminal symbols. In this CFG it is clear that a tree produced by $a \to b$ could never be a subtree of a tree produced by $b \to b$. However, unlike the 'match' operator, the 'subtree' operator cannot be entirely eliminated. Take our previous example and consider the rules $b \to c$ and $b \to b$. It is entirely possible that a tree produced by the rule $b \to c$ is a subtree of a tree produced by $b \to b$. On the other hand it is also possible that it is not a subtree of a tree produced by $b \to b$ as it may end with a tree produced by $b \to c$ instead.

This 'subtree' operator presents a problem with our previous proof. At the end of the previous section, we showed the problem with quantifying trees not part of the relations. We solved this problem by looking at 'extract' operators in the formula and used that to create a pair of structures it could not see the difference between. However, that trick of looking at the extract will no longer work, since the 'subtree' operator could work as an extract that could extract at an arbitrary depth an arbitrary child. The Spoiler could thus create a tree with some root node and all of its nodes represent pairs of trees from $U_1^{\boldsymbol{B}}$ and $U_2^{\boldsymbol{B}}$. These pairs represent a bijective function. The Duplicator is then forced to respond with a similar pairing but cannot since this is impossible to create in $\boldsymbol{A}$. It will either create a tree where at least one node is paired with multiple nodes or where at least one tree is not paired at all. If the Duplicator leaves out a single tree, the Spoiler will choose this tree from $\boldsymbol{A}$. Next the Duplicator will pick some tree from $\boldsymbol{B}$. Then the Spoiler will choose the tree in $\boldsymbol{B}$ that was paired with the tree just chosen by the Duplicator. The Duplicator will then choose some tree in $\boldsymbol{A}$. Finally, the Spoiler will win by constructing a pair from the two trees chosen in $\boldsymbol{B}$ and claiming it a subtree of the tree it chose at the very beginning of all pairs. The Duplicator cannot create such a tree because its counterparts in $\boldsymbol{A}$ were not a pair in tree it duplicated at the very beginning. Alternatively, assume the Duplicator created a tree where one tree is part of multiple pairs. The Spoiler will then first pick the node from $\boldsymbol{A}$ that has multiple pairs and then in subsequent turns will pick two other nodes it is paired with. Finally, it will construct these two pairs and claim that they are subtrees of the originally created tree. These constructed trees will not be possible to be created by the Duplicator in $\boldsymbol{B}$.

Note that this approach would not work without the 'subtree' operator because we can extract only up to a particular child. This means that we could not create a relation that contains one pair for each tree in a relation. This is because we could only ever access such a node by explicitly extracting it. Thus, if we know the largest child accessed, say the $c$-th child, we could simply make the structure larger so that it can no longer access all pairs. We would do this by having more than $c$ trees in each relation, which means the formula could only access the first $c$ pairs, but not all pairs.

### 5.2.3 Relational Subtree-Algebra

At the end of the previous section, we discussed how the proof that the count comparison query could not be expressed in the relational tree-algebra no longer holds when we add a subtree. In this informal proof, we constructed a tree which represented a bijection between two relations. This will also exactly be the approach we will take to show that the count comparison query, can indeed be expressed in the relational subtree-algebra. We will first define the new subtree operator, so that it can be used in a calculus. Then we shall give some translations that make it easier to define our formula. These translations will allow us to define our formula in second order logic where our second order

| $S$ | | |
|---|---|---|
| 1 | 2 | 3 |
| $r_{11}$ | $r_{12}$ | $r_{13}$ |
| $r_{21}$ | $r_{22}$ | $r_{23}$ |
| $\vdots$ | $\vdots$ | $\vdots$ |
| $r_{91}$ | $r_{92}$ | $r_{93}$ |

Table 5.1: An example of a relation with only tree values to be translated

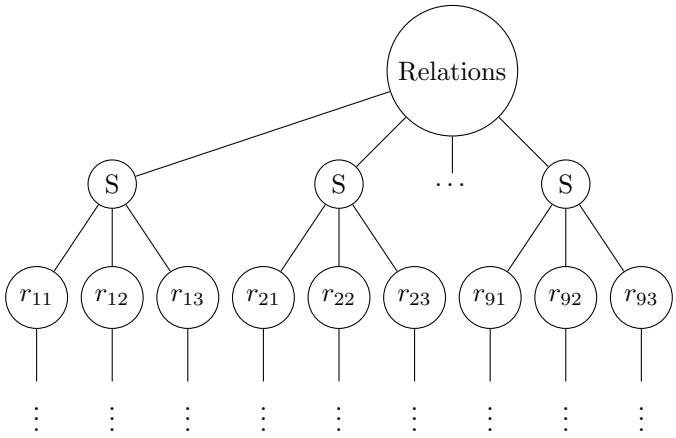



Figure 5.7: An example of how Table 5.1 could be translated to a tree

variables are relations over trees. Finally, we will give a formula that can be translated into a formula in our calculus, which returns the answer for the count comparison query.

### Subtree Predicate for Calculus

Before we can begin to use the power of the 'subtree' operator, we must define how it will be accessed in our calculus. This would be through a predicate '$subtree(t_i, t_j)$' where $t_i$ and $t_j$ are tree variables. The '$subtree(t_i, t_j)$' evaluates to true if 'subtree$_{i,j}$' would keep the row and $t_i$ was the tree in column $i$ and $t_j$ the tree in column $j$ of that row. This means that it evaluates to true if $t_i$ appears as a subtree in $t_j$. This predicate is then added to the previously defined calculus over trees. If we add this operator to the relational tree-calculus, we get the *relational subtree-calculus*. We shall sometimes shorten this to *RSC*.

### Second-Order Logic Translations

**Second-order relations** The first translation we shall introduce is the most fundamental one. This translation will allow us to simulate quantifying a second-

order variable, where the relation only contains trees. As an example, we have translated Table 5.1 into the tree in Figure 5.7. In Table 5.1 each value is a tree value, which is here represented by the label of its root. This is technically incorrect, but we shall use this for the sake of keeping this table and the translation somewhat compact. The translation of the relation would be done, as already described earlier, by defining a tree with some root label, with its children representing rows in a relationship. In our case we chose this root label is 'Relations'. It does not matter what the name is of this root label, but is best advised that this a label that does not show up elsewhere in any other tree. Next the children of this root node all have the label or the name of the variable. In our example this is '$S$' since we are representing a variable with the name $S$. Here again it is advised to choose a variable name that is not used as a label in a tree elsewhere. For each row we wish to have in our relation, we create a child with the label of the variable. Finally, each of these children get an amount of children equal to the arity of the variable. The $i$-th child will have as its entire subtree, the value of the tree in the $i$-th column. This is clear to see in our example where the first child of the first $S$ node is the root node of value of the first column in the first row of the $S$ we are trying to represent. We then assume that the rest of subtree underneath this node is then exactly tree in the first column of the first row of $S$.

**Containment in second-order relations** With the previous paragraph we have now created a way for us to essentially write $\exists S(\varphi)$ with $S$ a second-order variable and $\varphi$ a formula in RSC. However, we have not yet defined how to access whatever is contained within $S$. This will be done using the 'subtree' predicate together with a 'construct' predicate. Take example from Figure 5.7 and assume this tree was created by an existential quantifier like $\exists S$. Assume that $t_1, t_2$ and $t_3$ are all tree variables, we will then show how to define $S(t_1, t_2, t_3)$. This can be done by defining the following sentence:

$$\exists t_4(\mathit{construct\text{-}S\text{-}3}(t_4, t_1, t_2, t_3) \wedge \mathit{subtree}(t_4, S)) \tag{5.8}$$

In this formula we first quantify a new variable $t_4$ to put our new tree into. Then we construct a tree that should be a child of our tree variable $S$ and put it in $t_4$. Finally, we check that $t_4$ is indeed a subtree in $S$ somewhere. This also shows why it is important that $S$ as a label is not used elsewhere in any of the trees. If the label $S$ were used in other trees, it is possible that we could match with a subtree that is not a child of *Relations*. Assume we used the label $S$ in the trees elsewhere, but still wanted to guarantee that we only matched children of relations as our $S$. We could do this by adding $\exists t_5(\mathit{construct\text{-}Relations}\text{-}1(t_5, t_4))$ and then check that $t_5$ is a subtree of the tree $S$. This would then force the root node of $t_5$ to match with the root node of the tree variable $S$, since we assume this is a unique label not used in any other tree.

**Count Comparison Query Proof**

Finally, we shall now prove that we can express the count comparison query by using the second-order variables to express them. We could of course express them entirely in relational subtree-calculus, but this formula can be translated to that. On top of that, using second-order variables allows it to be somewhat easier to be understood. We shall start by giving the formula $\varphi_{CC}(C_1, C_2)$ and then slowly analyzing what each piece does. In this formula $R(t_1, t_2)$ still represents the pairs of subtrees of AST's that are similar.

$$\begin{aligned}\varphi_{CC}(C_1, C_2) \equiv \exists S\big(\forall x\big(R(C_1, x) \rightarrow \\ \exists y\,(S(x, y) \wedge R(C_2, y) \wedge \neg\exists z(S(x, z) \wedge z \neq y))\,\big)\wedge \\ \exists w\,(\forall q(\neg S(q, w) \wedge R(C_2, w)))\big) \qquad (5.9)\end{aligned}$$

Note that this does not directly solve the structure used to prove that the count comparison query could not be expressed using the RTC. However, on that structure we proved that a query $\varphi_{U_1\text{-}smaller}(u_1, u_2)$ could not be expressed and that the count comparison query could be used to solve it. This then proved that the count comparison query could not be expressed in the relational tree-calculus. Here we will propose a formula that can be translated so that it does solve the count comparison query. Take this formula and replace every reference to $T(C_1, x)$ with $x$ some value or some variable with $U_1(x)$ and replace every reference to $T(C_2, y)$ with $y$ some value or some variable with $U_2(y)$. The resulting formula is a query that for $\varphi_{U_1\text{-}smaller}(u_1, u_2)$. Alternatively we can use the same approach as in the previous proof to express $\varphi_{U_1\text{-}smaller}(u_1, u_2)$ using $\varphi_{CC}(C_1, C_2)$.

Equation 5.9 is clearly divide in two parts, each representing different concepts that the query is enforcing. The first part is enforcing that $S$ is injective. The second part is enforcing that $S$ is not surjective. The first part is everything quantified by $\forall x$. This part forces that $S$ must be injective by forcing that for every value of some domain, in this case all $x$'s in $R(c_1, x)$, it has some counterpart in $S$, namely $y$. On top of this it forces that it does not map one $x$ to two different values. It does this by stating that there is no such a $z$ where $z$ and $y$ are different and $S$ maps $x$ to $z$ on top of already mapping $x$ to $y$.

The second part is $\exists w\big(\forall q(\neg S(q, w) \wedge R(c_2, w))\big)$ which expresses that the function is not surjective. A function is surjective if for every value in the image there is at least one value in the domain. Thus, to check that a function is not surjective, we simply check that there exists some value $W$ in the image such that there is no value $q$ in the domain that $S(q, w)$ is true. To enforce that $w$ must come from the image and is not some random tree, we used $R(c_2, w)$.

We have now proven that there is indeed a formula that in second-order logic describes our count comparison query. We have also proven that all the aspects of second order logic here can be translated into relational subtree-calculus. This means that we have thus proven that we can express the count comparison query in the relational subtree-calculus. Note how this is only possible thanks to the addition of the 'subtree' operator. It also shows the power of the 'subtree' operator and its complexity. Thanks to the 'subtree' operator it is now possible to express queries in second-order logic, since $\forall R(\varphi)$ with $R$ a second-order variable and $\varphi$ some second order formula can be written as $\neg\exists R(\neg\varphi)$. Second-order logic is however much more computationally complex than first-order logic, which is the trade-off made for this expressive power.

## 5.3 Similarity

So far in all preceding sections, we have assumed the existence of some relation $R(C1, C2)$ that contains all pairs of similar patterns. However, this is one of the central parts of code pattern mining algorithm: finding these pairs of similar fragments. It would obviously be very convenient if we could simply define an entire algorithm with a single query. In the rest of this section we will show that this is indeed possible, depending on the definition of similarity, with the relational subtree-algebra. We shall show the relational subtree-algebra can be used to define FREQT and FREQTALS by defining pattern tree similarity. We shall also show that for similarities based on an intermediate language we cannot express their similarity in the relational subtree-calculus. On top of this, we shall show that for the API set and API sequence match we can express these in the RSC but that it can become a very complex formula quickly.

### 5.3.1 Pattern Tree Similarity

As discussed in Section 3.2, there are multiple types of similarity. The first type of similarity we can define using the relational subtree-calculus, is pattern tree similarity. This is particularly useful as it will allow us to define algorithms like FREQT and FREQTALS that use this similarity.

This type of similarity can be rather easily and naturally expressed with the 'subtree' operator. However, the issue is that the equality relation only checks that the labels and structure of the trees are the same. Thus, we could have two subtrees with the exact same labels and structure but in different locations in and the equality relation would consider them the same. The $R$ relation we are attempting to simulate however, would consider them two different fragments that are similar. This means that we somehow need to find something that can differentiate the two. One possible solution could be to check the path from the root nodes of those two trees and check if they are the same. If they are the same we would consider these trees to be the same and if they are different we would consider these trees to be different subtrees. This however

only works if we enforce the restriction that for each node, each of its children must have different labels. This was also mentioned when comparing examples in Section 3.1.1. However, if this restriction is not in place it is not guaranteed that method works. An alternative approach we could take is to generate a path from the root node, allow only one node to have two children and then check that each of these children are the root of a path that leads to the trees of the fragments.

As an example of this first approach, consider the AST in Figure 5.8a where we have highlighted a path from the root to the tree in Figure 5.8b. We can see here how we match exactly the given subtree at the base of our path and then have a chain of single child nodes above it.

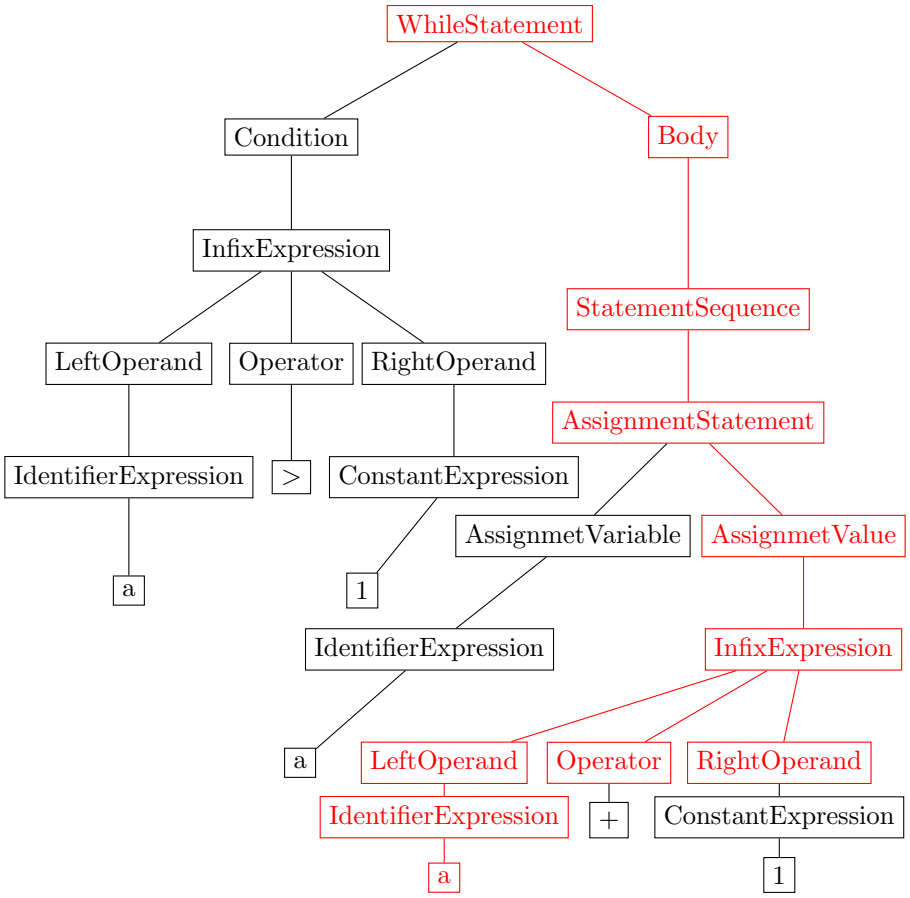
For the second approach we will essentially use the first approach, thus we shall start by defining the first approach in RSC. This means that we need to define a way to find a subtree that is a given tree and then a path all the way to a particular node. In the first approach this node would be the root node. We shall first give the formula and then dissect it to make it more understandable. Assume we call this formula $\varphi_{exact\text{-}path\text{-}a}(t_1, t_2)$ where $a$ is the label of the root node, $t_2$ the tree we are trying to find the path to this root for and $t_1$ a path from the tree to the root node. Also assume that there is a tree variable $ast$ which contains the entire AST of the program we are mining for similarities.

$$\begin{aligned}\varphi_{exact\text{-}path\text{-}a}(t_1, t_2) \equiv & match\text{-}a(t_1) \wedge subtree(t_1, ast) \wedge subtree(t_2, t_1) \\ & \varphi_{exact-base}(t_1, t_2) \wedge \varphi_{path}(t_1, t_2)\end{aligned} \tag{5.10}$$

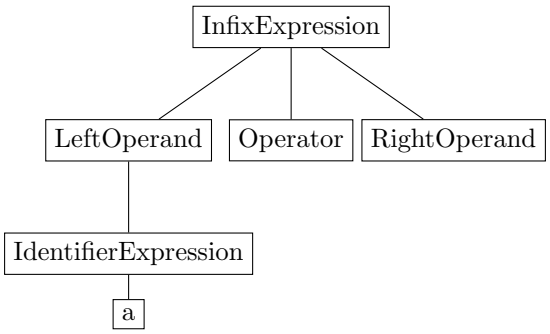
$$\begin{aligned}\varphi_{exact\text{-}base}(t_1, t_2) \equiv & \neg\exists x_1 \big(subtree(x_1, t_1) \wedge subtree(t_2, x_1) \wedge x_1 \neq t_2 \wedge \\ & \neg\exists y_1 (subtree(y_1, x_1) \wedge extract\text{-}1(t_2, y_1))\big)\end{aligned} \tag{5.11}$$

$$\begin{aligned}\varphi_{path}(t_1, t_2) \equiv & \forall x_2 \big((subtree(x_2, t_1) \wedge subtree(t_2, x_2) \wedge x \neq t_2) \rightarrow \\ & (\exists y_2 (extract\text{-}1(y, x)) \wedge \neg\exists z_2 (extract\text{-}2(z_2, x_2)))\big)\end{aligned} \tag{5.12}$$

It is clear that Equation 5.10 can be split into three pieces. The first piece is the simplest piece. $match\text{-}a(t_1) \wedge subtree(t_1, ast) \wedge subtree(t_2, t_1)$ simply states that $t_1$, the path we are looking for, must be a subtree of the AST we are trying to find it in. This AST is given by the constant $ast$. This piece also enforces that the root of the path and the AST must be the same, assuming the AST has the root label $a$. However, this rests on the assumption that the root label of the AST is unique. Should this not be the case, this could easily be solved by adding a new root label that is not present in $ast$ as a new root to the AST. Assume the label $b$ is not used in $ast$. We would then change the formula to be $\exists ast_{new}(construct\text{-}b\text{-}1(ast_{new}, ast) \wedge \varphi_{exact\text{-}path\text{-}b}(t_1, t_2))$ and replace the




(a) An example of an AST with a path to a subtree highlighted




(b) AST subtree we are attempting to find

Figure 5.8: The AST's of the tree with the path to the subtree highlighted and the subtree itself

reference to $ast$ in $\varphi_{exact\text{-}path\text{-}b}(t_1, t_2)$ with $ast_{new}$. Finally, it also states that $t_1$ must have the tree $t_2$ as a subtree in it, since we are looking for a path from the root to $t_2$.

The next piece of Equation 5.10 is Equation 5.11. In this piece we enforce that the path $t_1$ at its base can only have exactly $t_2$ and nothing more. If we did not enforce this, we would get many duplicate matches: one for each subtree underneath the root node of $t_2$ that contains at least $t_2$. To prevent this duplication we enforce that there is no subtree from the root of $t_2$ down that contains more than $t_2$. We do this by saying there is no tree that is a subtree of the path and contains $t_2$ as a subtree that is not exactly $t_2$. On top of this, we enforce that this subtree that cannot exist, cannot have another subtree within it that when extracting its first child gives $t_2$. This last part expresses that it must have the root of $t_2$ at its root. If this is not the case, then that means that there is something above $t_2$. If there is something above $t_2$, then it is possible to find a subtree within that $x_1$ with exactly $t_2$ and the node above it.

To make it more understandable what Equation 5.11 does, consider the subtree highlighted in Figure 5.8a. Here we see that if we added the 'ConstantExpression' node underneath the highlighted 'RightOperand' we would get another subtree. This subtree would also contain the tree of Figure 5.8b as a subtree. Thus, we would have multiple subtrees that have paths to the root that contain the requested subtree. To prevent this Equation 5.11, enforces that in the found path there is no subtree that contains the requested tree as a subtree, has no nodes of the path above it and is not exactly the requested subtree. It thus disallows that there is a subtree that is an expansion like the subtree where we added the 'ConstantExpression' node.

The last piece of Equation 5.10 is Equation 5.12. This piece enforces that every node above the root node of $t_2$ may only contain a single child, thus making it a path form the root to $t_2$. We do this by checking every subtree $x_2$ of the found tree $t_1$ that has $t_2$ as a subtree but is not $t_2$ itself. This means, in combination with the previous piece, that $x_2$ must have at least one node above the root node of $t_2$. Next we simply check that it then has a first child but not a second child with the 'extract' operator. Clearly if it has no second child it does not have a third child or higher, since it would then have a second child. This means that the root node of that $x_2$ only has a single child, making it a path to $t_2$.

This shows how Equation 5.10 gives a formula for how to determine a single path from the root node from a tree to some fragment $t_2$. Note that since the AST is a tree, this path is unique in the AST. Thus, if for every node in the AST each of its children have names different from each other, this path will suffice to uniquely determine one occurrence of $t_2$. A relation $Sim(C1, C2, P1, P2)$ could be defined where $C1$ and $C2$ are equal, but their paths are different. This could

then be used to create the similarity relation $R$. An issue however with this kind of similarity is that we cannot simply use $\pi_{1,2}(Sim)$, keeping only the $C1$ and $C2$ columns. This is because we are assuming set semantics, which would cause all the different occurrences of $C1$ and $C2$ to collapse into one. Because of this we would work with the $P1$ and $P2$ columns to keep different occurrences separate. Using $Sim$ instead of $R$ is rather simple. If you want to refer to all code fragments of a particular form, use the $C$ columns. If you need to refer to one particular code fragment in a specific location and the location of all its similar fragments, use the $P$ columns.

However, if not all children of the same node have different names, we cannot rely on the paths not being equal to paths leading to a different location in the tree. To solve this we proposed using a path that at one point splits in two and then continues on to the two similar fragments. Assume the label of the AST is $a$ and that it is unique. Also assume that the AST is contained in a constant $ast$. The formulas $\varphi_{fork\text{-}a}(t_1, t_2)$ then gives such a path $t_1$ for a particular fragment $t_2$. Again we shall first give the formula and then explain its pieces.

$$\begin{aligned}\varphi_{fork\text{-}a}(t_1, t_2) \equiv & \mathit{match}\text{-}a(t_1) \wedge \mathit{subtree}(t_1, ast) \wedge \\ & \exists x_3 \big( \mathit{subtree}(x_3, t_1) \wedge \varphi_{child\text{-}1\text{-}exact\text{-}path}(x_3, t_2) \wedge \\ & \varphi_{child\text{-}2\text{-}exact\text{-}path}(x_3, t_2) \wedge \neg \exists y_3 \left( \mathit{extract}\text{-}3(y_3, x_3) \right) \wedge \\ & \varphi_{path}(t_1, x_3) \wedge \varphi_{exact\text{-}base}(t_1, x_3) \big) \end{aligned} \tag{5.13}$$

$$\begin{aligned}\varphi_{child\text{-}i\text{-}exact\text{-}path}(x, t_2) \equiv & \exists y_4 \big( \mathit{extract}\text{-}i(y_4, x) \wedge \mathit{subtree}(t_2, t_1) \wedge \\ & \varphi_{exact\text{-}base}(y_4, t_2) \wedge \varphi_{path}(y_4, t_2) \big) \end{aligned} \tag{5.14}$$

The first piece of Equation 5.13 is $\mathit{match}\text{-}a(t_1) \wedge \mathit{subtree}(t_1, ast)$, which is similar to the first piece in Equation 5.10. It again enforces that the found path $t_1$ starts at the root node of the AST and that the path is part of the AST.

The second part of Equation 5.13 is the remaining part of the formula within $\exists x_3$. This section enforces that the root node of $x_3$ is the node where the path splits and then goes to the two fragments that are both $t_2$. It also enforces that the parts that are not part of the $t_2$ fragments or the node where the path forks, only have a single child. First the formula enforces that $x_3$ must be a subtree of the forked path $t_1$. Next it enforces that the first child must be a path like the ones we described in the previous equation. In a following paragraph we will explain how this formula works in more detail. Then it enforces the same for the second child, which also enforces that the root of $x_3$ is the point at which $t_1$ forks. Next, this part of the formula enforces that this node has no third child and thus that it only forks in two here. After that, the formula enforces that starting from this fork in the path, all nodes above it only contain a single child. Finally, the formula makes sure that $t_1$ contains the fork in the road subtree $x_3$

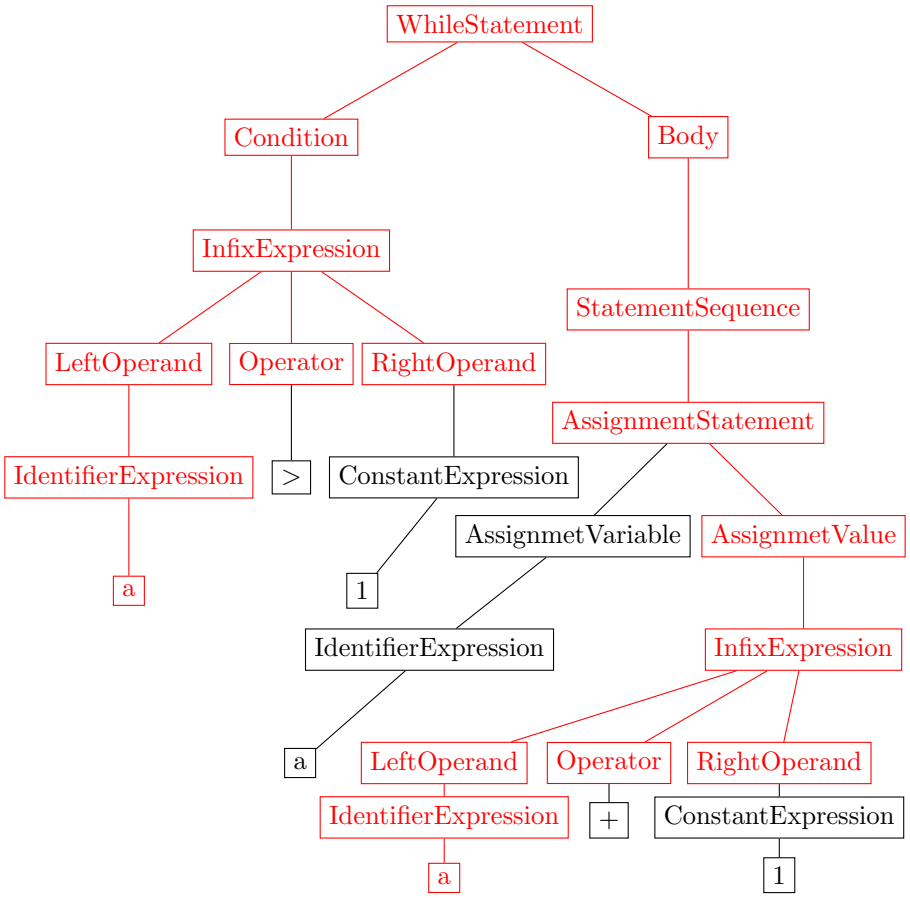



Figure 5.9: An example of an AST with a forked path to two subtrees highlighted

exactly and that no more nodes are added in $t_1$ to underneath $x_3$. This means that we have enforced that we have a single path to a fork in a road, which then has paths which lead to the two $t_2$'s. Starting from this fork in the path there cannot be any other forks or possible paths in $t_2$ other than the two previously mentioned ones. This is enforced by the last part of this piece of the formula.

If we consider the example of Figure 5.9 we can understand what the second part of the equation does. In this example we shall skip over the children of the fork since this will be explained in the following paragraphs. Firstly the equation enforces that everything above the fork, in this case at the 'WhileStatement' node, is a single path with $\varphi_{path}(t_1, x_3)$. In this case however it is rather trivial since there are no nodes above the 'WhileStatement' node. Next it enforces that there is no third child at our fork, which is again trivial in this case, as the'WhileStatement' node only has two children. Lastly with $\varphi_{exact\text{-}base}(t_1, x_3)$ it enforces that entire subtree within the path starting from the 'WhileStatement' node is exactly the subtree of the path underneath the 'WhileStatement' node. This may seem rather self-fulfilling, however it is important because if this were not the case it is possible that at the 'WhileStatement' node the path forks in three. This would be possible since previously we were only considering one subtree of the path where a fork occurs. However, if the path actually forks in three, then there are three subtrees that we could have been considering where all the proceeding statements were correct. $\varphi_{exact\text{-}base}(t_1, x_3)$ enforces that this is the only such subtree.

The final part of Equation 5.13 that needs to be explained is Equation 5.14. This part of the formula enforces that the $i$-th child of the passed $t_1$ is a path from that node to $t_2$, just like in Equation 5.10. However, in this case we do not have to enforce that $t_1$ is part of the AST, since this is already done by Equation 5.13. We also do not need to enforce a particular label is the root node since this is already enforced by passing $x_3$ and extracting $y_4$ from it.

To give an example of Equation 5.14, consider Figure 5.9. In this case the fork happens at the root of the tree. Equation 5.14 enforces that the entire subtree of the path starting from the node 'Condition' is one similar to the paths described in Equation 5.10. It also does the same for the entire subtree of the path starting from the 'Body' node.

This shows that with Equation 5.13 we can define in the relational subtree-calculus the similarity query. It also shows that we can do this, even if the different children of one node do not have different labels. This could computationally make the problem quicker to solve, since the AST could become more compact, but the more complex formula might make it slower to execute, even with the smaller tree. This clearly shows the trade-off between the two approaches. Another issue with this approach, is that like the previous approach, it cannot simply be output to $R$. The reason is again that set semantics in

combination with our similarity would cause all occurrences of a pair of similar fragments to collapse into a single pair. Thus, we are again stuck with using a different relation: $Sim(C1, C2, P)$. In $Sim$ the fragments $C1$ and $C2$ are equal but the path $P$ clearly shows that they are both reached by a path that splits at some point. Again here we thus solve the problem of set semantics by adding a forked path that will be unique for each pair. This is also relatively simple to replace $R$ with. If we simply need to refer to the similar pairs but do not care about their locations we use $C1$ and $C2$. If we care about the location of pair, we use column $P$. Again, for this similarity using both $C1$ and $C2$ is rather useless since they will always be the same.

### 5.3.2 Tree-mining algorithms

Now that we have defined how we could extract the pairs of similar trees, we can also attempt to describe an entire algorithm. If the size of $ast$ is know, this task is easy: we take Equation 5.1 and replace $R(c, c_j)$ with $\varphi_{fork\text{-}a}(c_j, c)$. To get all subtrees that are $s$-frequent, we simply need to take $|ast| * s$ where $|ast|$ is the amount of nodes in $ast$. This then gives us a query in the relational subtree-calculus that immediately gives us the output of the FREQT algorithm.

Moving from FREQT to FREQTALS, would simply require us to translate the particular constraints that we wish to impose upon our fragments into relational subtree-calculus. We would then add these restrictions to the formula of FREQT by adding them with conjunctions to the formula of FREQT. The only aspect that is more difficult to add to FREQT in the relational subtree-calculus is the maximal subtree mining. This could however be done by first executing the constrained FREQT. After that, similar to Equation 5.11, we would check all trees that originate from the same root as the input tree and have the input tree as a subtree. Next, a binary relation could be created, mapping all paths of the original code fragment to the paths of the new code fragment. If such a relation is injective then the largest code fragments with the same original code fragments should be chosen. Selecting the largest code fragments can be done by expressing that there is not another code fragment that contains this code fragment as a subtree of it.

### 5.3.3 Other types of similarity

So far we have only discussed the pattern tree similarity, but not any type of similarity based on an intermediate language. This is because in relational subtree-calculus there are no tools to rewrite subtrees at an arbitrary location. This could be created by adding additional constructs such as a 'rewrite-one' and 'rewrite-all' operators, similar to those of the relational meta-algebra. F. Neven et al. discussed in the relational meta-algebra how the rewrite operators were not derived operators[NVVV99]. They also demonstrated that their 'submatch' operator, which is somewhat similar to our 'subtree' operator, could be written with their 'extract' operator. Their 'extract' operator however extracts at an

arbitrary depth. Given the similarities between the relational meta-algebra and the relational subtree-algebra, it is thus likely that the rewrite operators could not be expressed in the relational subtree-calculus. This would mean that to support this type of similarity we would need to add even more operators to our language. This would make the language more expressive and potentially adds more computational complexity. Additionally, this shows how using an intermediate language, can potentially make a query more complex when trying to reduce the complexity.

It should be straightforward, once a set of rewrite operators are introduced to write a query that mines using an intermediate language. The formula would first rewrite the tree into the intermediate language and then replace the references to $ast$ in our previous equations with the rewritten tree. Alternatively, it could also be used after pattern mining to consider code fragments the similar if they can be rewritten to the same fragment in the intermediate language. If we use Equation 5.13 for this, we would then also need to allow an additional parameter $t_3$ which would be subtree found at the end of the second path in the fork. With this new operator we could then find all pairs of paths linking up two trees that are similar in the intermediate language.

Similarly the API set similarity en API sequence similarity have not been discussed so far. This is because they are not used in any of the algorithms we discussed. However, these can be defined in the relational subtree-calculus. For API set similarity, it would simply be a case of checking that every subtree of one code pattern with as its root the label indicating a function statement and its name child is also a subtree in the other code pattern. For API sequence similarity, we would need to do the same but also match the control structures they are in. We could to this by comparing all pairs of function calls of the two code fragments and the control structures in their path above them to the root node of the fragment. For each of the control structure labels, we would then check that its other pair in the other fragment there is a similar control structure with the same control structures below it. However, we would also need to enforce the order in which these control structures are applied. We would do this by enforcing pairs of control structures are above and below each other with no other control structure in between. This would then be enforced by a binary second-order variable, containing all these pairs. However, it is clear that this very quickly becomes a very complex formula. Thus, we shall only give the rough description of how to structure the formula rather than fully write it out. We must acknowledge that this also leaves some gaps where it might turn out to not be possible to define. However, we believe it will simply be very complex, but not impossible.

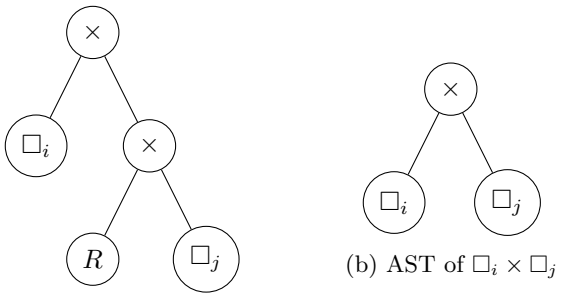
## 5.4 Relation with Other Languages

Now that we have discussed how the relational subtree-calculus can express many of our desired queries, we will investigate how it compares to other languages we have discussed. We will start by comparing it to the relational meta-algebra and relational meta-calculus. After this we will compare it to the Meta-SQL, a language with a very similar goal to our relational subtree-algebra.

### 5.4.1 Relation with Relational Meta-Algebra

In the previous sections we have proven that certain queries like the ones discussed in Section 5.1 and Section 5.2 can be entirely written in the relational subtree-calculus. When looking at the relational subtree-calculus, it seems somewhat similar to the relational meta-calculus, which is the calculus of the relational meta-algebra. This in turn seems to point that the relational meta-calculus could also be used to mine frequent patterns in the relational algebra. One issue however is that in the relational algebra, we typically do not have very large formulas, unlike in code where we often have very large AST's. This means that we are more likely to want to search for common patterns across the different formulas rather than within a single formula. This could be resolved by creating one formula that is a Cartesian product of all the formulas we would like to inspect. Defining this formula is difficult in the relational calculus and would have some side effects. For example, it would create an extra pattern of these chained Cartesian products. A formula to create this Cartesian product could relatively easily be written for a fixed amount of elements. However, creating such a formula becomes an issue when the amount of relational algebra formulas we wish to compare becomes of an arbitrary size. In the following paragraphs we shall explain this and other issues when trying to mine relational algebra patterns using the relational meta-algebra.

First we shall discuss the parts that are similar. When looking at the operators, there are some operators that are clearly similar. As an example, both contain an 'construct' operator. In the relational meta-algebra, a construct operator is expressed using a 'wrap', a 'rewrite-one', an 'extract', a 'rewrite-all', a Cartesian product and a projection [NVVV99]. This means that even though it is not a primitive operator, it can still be expressed in the relational meta-algebra and thus the relational meta-calculus. This construct can then be used to create a formula in which certain parts of it are replaced by other values of relational algebra formulas. The usage of this operator is similar to the construct operator of the relational tree-algebra. However, in the relational tree-calculus and relational tree-algebra we can only construct a single layer at a time. If we compare this with the AST of a relational algebra expression created by the construct of the relational meta-algebra we see that it can produce an entire subtree at once, compared to a single new layer. This can be seen when looking at the AST of $\square_i \times (R \times \square_j)$ in Figure 5.10a. However, if we limit the construction rules to only contain a single operator and operands referencing




(a) AST of $\square_i \times (R \times \square_j)$

(b) AST of $\square_i \times \square_j$

Figure 5.10: AST's of construct formulas in relational meta-algebra

other columns, we can only do a single layer. An example of this can be seen in Figure 5.10b, where we translate $\square_i \times \square_j$. This means that the construct operators would become equivalent, if we restricted the 'construct' operator of the RTA to the CFG of the relational algebra. However, clearly the 'construct' operator of the RMA could not create an AST where a node has more than two children. We can however simulate the 'construct' operator of the RMA in the RTA if the formula is known beforehand, by simply chaining construct operators to make the AST, one layer at a time.

The next operator of the relational tree-algebra to consider is the 'extract' operator. This operator extracts a given child from the root node of a given tree. In the relational meta-algebra we also have an extract operator. This operator however, extracts all subformulas of a particular arity at an arbitrary depth. Although describing the 'extract' operator from the relational tree-algebra seems hard in the relational meta-algebra, in the relational meta-calculus we can come closer. Since we know there is a derived 'construct' operator in the relational meta-algebra, there must also be one in the relational meta-calculus, because they are proven to be equivalent [NVVV99]. We could then extract the first child by constructing a large disjunction of the possible 'construct' operators that could generate a formula with at least one or two children. The sets of formulas our construct disjunction must generate depends on whether we are extracting the first or second child. Since there are only a handful of operators, we could easily write all combinations of $\square_i \times \square_j$, $\pi(\square_i)$. Assume the 'construct' operator in the RMC was $construct(result, \square_i, \square_j, formula)$ where $formula$ represents the formula with the $\square$'s and $\square_i, \square_j$ represent the variable containing their values and $result$ is the newly constructed formula. If we then wanted to extract the first child as $c$ from the expression $e$ we could write $\exists c_2(construct(e, c, c2, formula))$ for all formulas. However, while there may be a limited amount of RA operators, these can have an infinite amount of parameters. Operators like the projection can have any number in them up to the size of the relation. The size of a relation can be made arbitrarily large with Carte-

sian products. Similarly, the selection operator would have an infinite amount of possible labels. This shows that it is thus unlikely that the 'extract' operator of the relational tree-algebra could be written in the relational meta-algebra.

Next there is the 'match' operator of the relational tree-algebra. It is easy to assume that this operator can be written relatively easily in the relational meta-calculus, using the 'construct' operator. All we need to do is to simply construct a formula that uses the same operator and then has a single or two children in its AST, depending on the RA operator in the root. We then check that the constructed formula with the one or two children of the original formula is the same as the original formula. However, this clearly shows the issue with this approach: we rely on an extract operator, which we have shown to not be likely to be possible. This means that if the 'extract' operator of the relational tree-algebra is possible to be written in the relational meta-algebra, the same will go for the 'match' operator. Note that this is only possible since we know the CFG of the relational algebra and can use that.

Now that all operators of the relational tree-algebra have been discussed, we shall consider those of the relational subtree-algebra. The only extra operator is the 'subtree' operator, that we have shown gives a large amount of additional strength. This operator cannot be written in the relational meta-algebra, even though there is the similar 'submatch' operator. The 'submatch' operator however checks for the maximal subtree underneath a particular node. The 'subtree' operator on the other hand checks for a subtree, but this is not required to the maximal subtree for a node. This could be attempted to be solved by writing a set of rewrite rules that remove all the nodes underneath a certain node of a tree and insert a new placeholder leaf node. This however cannot work as it would require an infinite amount of rewrite rules. Note the similarity to how the 'extract' operator could not be expressed due to an infinite amount of labels the nodes could have. On top of this the size of the tree underneath a particular node could be arbitrarily large and the relational meta-algebra cannot rewrite arbitrarily trees. That is to say for a given rewrite we can always find a formula that is larger than it can rewrite to a single node.

Finally there is the 'rewrite' operators of the relational meta-algebra that do not have a counterpart in the relational subtree-algebra. As already discussed in Section 5.3.3, adding these would add a considerable amount of expressive power to the language. It is likely this is not possible to be written in the relational subtree-calculus, due to the ability to quantify the tree that will be rewritten. Once this is possible, we cannot simply look at the tree that we are replacing and extract an according amount of levels down to reach what would be the leaves of the subtree we are replacing. This is because this amount of levels deep to extract could be arbitrarily large. However, this is for the 'rewrite' operators as they would be usefully defined for a tree. If we consider the ones defined in the relational meta-algebra, we see that the leaf nodes of the tree

being replaced are expected to be leaf nodes for the tree we are replacing it in. This after filling in nodes of the type $\square_i$ with the tree value found in column $i$. This makes the problem considerably easier, since we will never have to go to the leafs of these trees that we are replacing and extract the trees underneath them to somehow reattach to the new tree. If we allow the tree to be replaced to be quantified, we need to ensure that this is the maximal subtree starting from the root node of the tree to be replaced in the tree to replace it in. This formula would likely be something similar to Equation 5.11. If we allow the tree that it will be replaced with to also be quantified, we need to confirm somehow that its leafs are also leafs of a tree and not some incomplete tree. This would be rather difficult, again due to the infinite amount of possible labels. If we restrict our calculus to only quantify trees to the CFG of relational algebra, the quantified trees will never be incomplete.

Another difficult part of defining the 'rewrite' operators is attempting to keep the rest of the tree the same and attaching the new subtree in the parts that need to be replaced. This is because removing a particular subtree is not easily defined in the relational subtree-algebra. On top of this attaching something at a fixed location would require following a chain of extract operators from the root to the subtree to be replaced in the new tree. This of course cannot be written in the relational subtree-calculus, since it may be at an arbitrary depth.

The preceding paragraphs have thus shown that it is likely not possible to do pattern mining on relational algebra expressions with the relational meta-algebra. This seems contrary to what one might expect, given the similarity between it and the relational subtree-algebra. However, as has been shown in the previous paragraphs, they cannot be easily translated into each other. In the previous paragraph we mostly discussed how the operators of the relational subtree-algebra could not be expressed in the relational meta algebra. However, the other way around also holds due to the lack of the knowledge of the arity of relations and formulas. The reason for the conclusion that pattern mining is unlikely to be possible, comes from the fact that the 'subtree' operator cannot be described in the relational meta-calculus. This operator plays an integral role in making pattern mining possible in the relational subtree-calculus.

### 5.4.2 Relation with Meta-SQL

Meta-SQL is the one other language that we have discussed that take the same approach to code querying as the relational subtree-algebra: it queries AST's. However, since the Meta-SQL is SQL with added functionality and the RSA and RSC are algebraic and logical model, our comparison will be less detailed than with the RMA and RMC.

As previously mentioned, the basic idea of both the Meta-SQL and the RSA is very similar. Both of these approaches store the queries as an AST and

then attempt to query and modify these through their operators. However, we have also mentioned that XSLT, which is used in Meta-SQL to query the AST's, is Turing-complete. This means that it could simulate any operators of the relational subtree-algebra. The reverse however cannot be said of the RSA, as we have previously already mentioned how a rewrite operator cannot be expressed in it. This means that XSLT clearly offers more power when it comes to querying AST's.

It is also clear that Meta-SQL has operators that the RSA and RSC do not have, such as the EVAL and UEVAL functions. These could also not be emulated in the RSA and RSC. However, these functions do not make sense in the RSC and RSA, as they are intended to mine patterns form source code. If we expected it to be able to evaluate any arbitrary program in an arbitrary language, we would also somehow need to give meaning to these AST. This in turn would require us to understand every AST for every programming language. This is clearly not our intention. If we restrict the Meta-SQL to only being used to query arbitrary code, rather than SQL expressions, we would only allow UEVAL and EVAL of AST's representing Meta-SQL queries. However, this is still much more powerful than what the RSA and RSC have, since it allows us to define functions that recurse an arbitrary amount. Thus, the EVAL and UEVAL still provide a lot of expressive power, even when the Meta-SQL is used to mine code patterns in AST's, since XSLT can create any arbitrary Meta-SQL expression to be evaluated with the EVAL and UEVAL functions.

This means that even though the RSA and the Meta-SQL take a similar approach on how to query code, the Meta-SQL is more powerful. This is mainly caused by the Turing-completeness of XSLT and the power of the UEVAL and EVAL functions. However, as already noted in Section 4.1.3 the Turing-completeness of the XSLT is also its weakness as querying may go on forever.

# Chapter 6

# Conclusion

In the previous chapter we defined the relational subtree-calculus and showed its expressive power. Firstly, we showed that it was strong enough to define the count comparison query. The reason that this could be achieved was thanks to the 'subtree' operator added in the relational subtree-algebra to the relational tree-algebra. The reason for this is that it can consider subtrees at an arbitrary location, rather than simply at a specific child. We abused this fact to allow us to create a second order variable, which allowed us to express the count comparison query. Following this proof we also proved that pattern tree similarity can be expressed with the relational subtree-calculus. This then lead us to given a description of how someone would go about defining FREQT and FREQTALS in the relational subtree-calculus. We then discussed how it likely not express the similarities using intermediate languages. We also considered how it would be possible to express the API set and API sequence similarities. Finally, we compared the relational subtree-calculus to the relational meta-calculus and Meta-SQL. For the relational meta-algebra we found that it could not simulate the relational tree-calculus. We also found that the relational subtree-calculus could not express the relational meta-algebra, not even without eval. This lead us to the conclusion that pattern mining is likely not possible in the relational meta-algebra. Finally, we compared the relational subtree-algebra to Meta-SQL. Here we found that, while they have a similar approach too storing and querying their data, Meta-SQL is vastly more powerful, which is both its strength and one of its issues.

However, many of the conclusions do not come without their caveats, which lead to future research questions. A first of these caveats in our result is that almost all our results about what the relational subtree-calculus can do, cannot be extended to the relational subtree-algebra. This is because we have not defined a safe variant of the relational subtree-calculus. This task we would likely look towards the definition of the safe relational meta-calculus for their approach. The reason for this is that both do not simply deal with active domain semantics, since both have operators that can produce values outside

the active domain. We would then have to consider whether our results also extend to the safe relational subtree-calculus. This was however not done in this thesis, simply due to a lack of time. We would have preferred to guarantee that the relational subtree-algebra can or cannot express the code pattern mining queries. However, we thought it would be more useful if this result could at least be fully achieved for the relational subtree-calculus, rather than only be partial achieved by both.

Another caveat is that there are some results of what the relational subtree algebra can say, where we have only described intuitively how to do it. This means that while we have things are likely to be definable in the relational subtree-algebra, we cannot say this for sure, since we have yet to fully write them out. This is the case for the results of being able to define FREQTALS, as well as for being able to define API set and API sequence similarities. If we had had more time, we would have tried to at least properly write out the API set similarity. We would do this since it would allow us to showcase the versatility that the relational subtree-calculus offers. It would also allow us to easily define a code pattern query using a different similarity than pattern tree similarity.

Another issue lies with the rewrite operators. We mentioned that to write any intermediate language similarity we would need rewrite operators. We also claim that it will probably add more expressive power. However, we do not ever define such operators, nor do we prove that some query cannot be expressed by the relational subtree-algebra. We would have liked to have explored the potential added power of additional rewrite operators and compared their power to those of the relational algebra. However, due to a lack of time, this was not further explored, since it adds a lot of work that contributes a relatively small amount to our research question. It would have allowed us to state that more kinds of code pattern mining are possible. However, this is of less importance compared to whether or not any code pattern mining is possible in the first place.

Finally we would like to summarize our most important contributions:

- We introduced the relational subtree algebra and the relational subtree calculus
- We proved that the subtree operator makes it possible to define the count comparison query
- We showed that the relational subtree-calculus allows us to use a query language to do code pattern mining

## 6.1 Future Research

As discussed in the previous section there are a few future research questions and opportunities that follow from this thesis. These mostly come from further

investigation where this research stopped due to time constraints.

**Safe relational subtree-calculus** A first important research path to go down, is that of the safe relational subtree-calculus. The safe relational subtree-calculus should be defined and be proven to be equivalent to the relational subtree-algebra. It can than be used to see if the results we found in with regard to what can be expressed with the relational subtree-calculus still hold if we restrict it to its safe counterpart. If this is the case then we can simply apply the findings of this paper to the relational subtree-algebra, which would give us a way to execute these queries we have defined. Should it not be possible to express our results in the safe RSA, it should be proven that these queries cannot be expressed at all. It should then also be researched what kind of additional operator would make these queries expressible.

**Intuitive algorithm descriptions** There are a few intuitive descriptions we have given of algorithms. However, these are relatively high level and do not account for possible problems that could be encountered when attempting to write them down. We believe that these descriptions should lead to actual formula that express these concepts, but a lot of the constructs used make intuitive sense but might have some complications when actually writing them down. Thus, it would be interesting to actually write these out, to prove that they can be written in the relational tree algebra. This would also lead to new tools being developed for the relational subtree-calculus, which would make it easier in the future to prove other concepts in it.

**Rewrite operators** A final interesting piece of future research are the rewrite operators. These operators are likely to add additional expressive power to the relational subtree-algebra, but it is unclear how much. Thus, it would be a good line of research to prove first that the translations into an intermediate language cannot be expressed by the relational subtree algebra. Then it would be interesting to see that it now suddenly does become possible with the rewrite operators. It would also be interesting to see what other queries suddenly become possible when we add the relational tree algebra.

**Complexity of the relational subtree-algebra** We have so far only considered the expressive power of the relational subtree-calculus. However, typically, expressive power comes with a trade-off in computational complexity. Thus, an interesting line of research would be to determine exactly how complex certain operators are. Especially the 'subtree' operator seems a prime candidate, since it adds a lot of expressive power.

## 6.2 Reflection

As a final section to this thesis, I would like to add some person reflection on how the thesis went and what lessons I have learned from it. To start off with how it

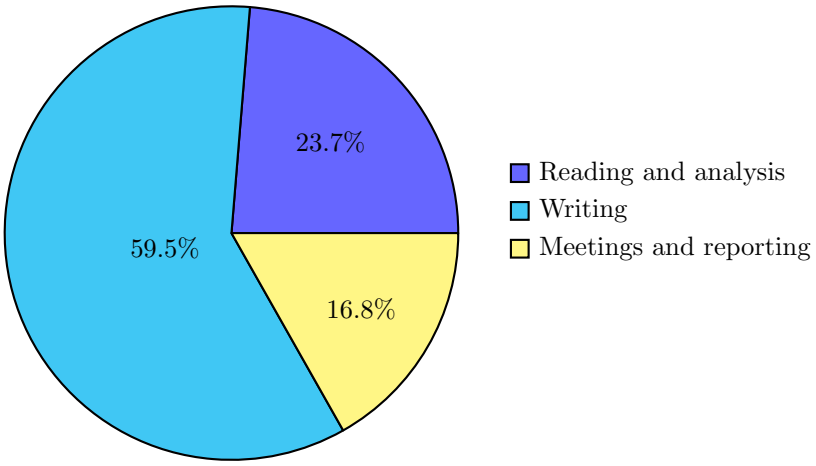



Figure 6.1: A pie chart how time was spent during the thesis

went, consider the pie chart in Figure 6.1. This chart contains a breakdown of on what the time was spent. It should be noted that the reading and analysis section contains all the reading but only part of the analysis. This is because I included only the analysis that I recorded as strictly analysis. However, while writing this thesis it was very common that I would start to write a proof, but would then find some issue with it. This would then cause me to stop writing and actually start attempting to solve this problem. However, all this time spent attempting to fix these problems were recorded under 'Writing'. Additionally, a fair amount of time from the meetings should also be categorized under analysis. I would often come to professor Van den Bussche with my problems I encountered during my research, and we would work through them together. For this I am very thankful, as it helped me out of a lot of difficult problems or put me on the right track to the solution.

During this thesis I learned a lot about finite model theory, beyond the shot extract was given in this thesis. I also learned things to consider when defining a query language. I also gained more of an intuition for what could affect the expressive power of a language and what would be a useful operator to add.

Something else that I learned during this thesis, is that I need to make sure to make things more concrete earlier. The reason for this is that a lot of the early time was spent reading without really knowing what direction I would exactly take this in. In the end I attempted to prove my research question by designing a new language, inspired by the relational meta-algebra. However, if I had made this concrete decision earlier on in the research, I could have potentially looked in more detail at certain aspects of the language. Now however, I spent a lot of time reading and considering many alternative approaches, rather than actively

working on a concrete path to answering the research question.

# Bibliography


[AAK+02] Tatsuya Asai, Kenji Abe, Shinji Kawasoe, Hiroki Arimura, Hiroshi Sakamoto, and Setsuo Arikawa. Efficient substructure discovery from large semi-structured data, 2002. Copyright - Copyright Society for Industrial and Applied Mathematics 2002; Document feature - Diagrams; Graphs; ; Last updated - 2012-03-26.

[C+72] Edgar F Codd et al. *Relational completeness of data base sublanguages*. Citeseer, 1972.

[CG85] S. Ceri and G. Gottlob. Translating sql into relational algebra: Optimization, semantics, and equivalence of sql queries. *IEEE Transactions on Software Engineering*, SE-11(4):324–345, 1985.

[dBVV05] Jan Van den Bussche, Stijn Vansummeren, and Gottfried Vossen. Towards practical meta-querying. *Inf. Syst.*, 30(4):317–332, 2005.

[Kay17] Michael Kay. XSL Transformations (XSLT) Version 3.0. Recommendation, W3C, June 2017. `https://www.w3.org/TR/2017/REC-xslt-30-20170608/`. Latest version available at `https://www.w3.org/TR/xslt-30/`.

[Kep04] Stephan Kepser. A simple proof for the turing-completeness of XSLT and xquery. In *Proceedings of the Extreme Markup Languages® 2004 Conference, 2-6 August 2004, Montréal, Quebec, Canada*, 2004.

[Lar21] Michael Larabel. Linux 5.12 coming in at around 28.8 million lines, amdgpu driver closing in on 3 million. `https://www.phoronix.com/scan.php?page=news_item&px=Linux-5.12-rc1-Code-Size`, March 2021. Accessed: 2021-06-08.

[MCJ20] Rohan Mukherjee, Swarat Chaudhuri, and Chris Jermaine. Searching a database of source codes using contextualized code search. *Proc. VLDB Endow.*, 13(10):1765–1778, June 2020.

[NVVV99] Frank Neven, Jan Van den Bussche, Dirk Van Gucht, and Gottfried Vossen. Typed query languages for databases containing queries. *Information Systems*, 24(7):569–595, 1999.

[PL16] Rachel Potvin and Josh Levenberg. Why google stores billions of lines of code in a single repository. *Commun. ACM*, 59(7):78–87, June 2016.

[PNM+19] Hoang Son Pham, Siegfried Nijssen, Kim Mens, Dario Di Nucci, Tim Molderez, Coen De Roover, Johan Fabry, and Vadim Zaytsev. Mining patterns in source code using tree mining algorithms. In Petra Kralj Novak, Tomislav Šmuc, and Sašo Džeroski, editors, *Discovery Science*, pages 471–480, Cham, 2019. Springer International Publishing.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I*, volume 14 of *Principles of computer science series*. Computer Science Press, 1988.