Alexander Taylor

A Purely Functional Approach to Graph Queries on a Database

Computer Science Tripos – Part II

St John's College

March 30, 2018

Proforma

Name:	**Alexander Taylor**
College:	**St John's College**
Project Title:	**A Purely Functional Approach to Graph Queries on a Database**
Examination:	**Computer Science Tripos – Part II, June 2018**
Word Count:	
Project Originator:	Adapted from a project proposal by Dr Eiko Yoneki
Supervisor:	Dr Timothy Jones

Original Aims of the Project

The initial aim of the project was to produce a simple graph database system which wrapped over an SQL database, to expose an API treating the database as a purely functional datastructure. Operations were to be achieved in a monadic fashion, according to well defined DSL semantics. The success criteria were that a user could write a composable query with the DSL and receive correct results with appropriate error checking.

Work Completed

All success criteria were met and exceeded. I also implemented several extensions such as adding write functionality and and a collection of bespoke back-ends using the LMDB key-value store. These new back-ends were evaluated by comparing their against the compiled SQL queries generated by the SQL back-end.

Special Difficulties

None.

Declaration

I, Alexander Taylor of St John's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed [signature]

Date [date]

Contents

List of Figures

Acknowledgements

Acknowledgements
    storage space of views - same as a write ahead log?
references between sections
urls to footnotes + biblio
pink book, software licenses
requirements analysis
starting point
rework of performance test analysis

Chapter 1

Introduction

In recent years, alternatives to the relational database model employed by SQL are receiving increasing attention. One such model for data is that of graph databases, which allow queries to pattern match over the graph formed by relations between objects in the the database. Another trend in the industry is a move towards languages with purely functional, strongly typed features. Fields which require the ability to quickly produce non-trivial, correct code have seen the use of high level languages such as Haskell, Ocaml, and Scala progress from a research interest to a serious paradigm. In these use cases, such as automated trading and safety critical systems, an advanced type system is used to make stronger guarantees about the runtime behaviour of the program and enhances the ability to find bugs at compile time than in more traditional imperative languages.

As the usage of functional languages for real-world code grows, there is an increasing need to integrate common patterns into the functional paradigm. Access to databases in one such example that requires solving. Frequently, when a functional program requires access to a database, such access is implemented ad-hoc for the specific usage case. For example, a program may be set up to read trade data from a database in a finance firm. This approach reduces generality and limits the kinds of abstractions that may be used. For example, the above usage case might only support specific schema required for the types representing the kinds trades that might appear; the wrapper used for the trading system could not be used to access a database containing a social network. In addition to this, while read only databases are trivially purely functional data-structures, it is harder to achieve referential transparency when writes are included into the database specification. This is because most database systems use imperative semantics to best make use of the hardware upon which they are implemented.

There has been some exploration into generalised libraries and wrappers for accessing databases in a purely functional way. These may allow a means to construct SQL queries in a type-safe manner (https://hackage.haskell.org/package/haskelldb), or provide manipulation of the databases state in a referentially transparent manner (http://www.bcs.org/upload/pdf/ewic˙dp95˙paper14.pdf) . However, they are typically aimed at relational (SQL) databases and do not allow safe generalisation over the types stored in the database.

In this dissertation, I introduce a general, performant, graph database system, written in Scala, that exposes its underlying graph as a purely functional data-structure.

Furthermore, it allows for tighter integration into the client application by allowing the application programmer to store their own types in the database, hence reducing the need for marshalling and unmarshalling. Queries constructed using the library are type-checked by the Scala compiler at compile-time, and execute according to a concrete semantics. In addition to these features, the system is constructed in a modular fashion, allowing for the back-end implementation technologies to be interchanged and the domain specific query language to be extended. My initial goals were to build a read-only database with these features and a single back-end which translates the submitted queries to SQL. However, I have significantly exceeded these goals by adding referentially transparent write operations to the database and writing several variations on a back-end which stores the graph using the LMDB key-value datastore.

Chapter 2

Preparation

This chapter discusses the early stages of the project. Once I had settled on a definition of type-safety and settled on using the Scala language for the project, I carried out a brief survey of graph database technologies to construct the language of queries that I intended to implement. Following this, I devised a set of semantics and constructed a system architecture.

2.1 The Scala Programming Language

I quickly decided to use Scala for this project. Scala is a JVM based object oriented and functional language. It provides a very advanced type system, allowing for complex abstractions to be built safely. Furthermore, due to its inter-operability with Java, the latter's libraries can be called from Scala, giving an advantage over other comparable languages such as Haskell and Ocaml, especially when there is a need to take advantage of low-level functionality. Finally, I was also familiar with some of Scala's more advanced features from previous experience.

2.2 Definition of Type-Safety

When we talk about type safety, we mean to say that a correctly typed program will not produce run-time errors of a given class. Breaches of type safety in typical programs include exceptions (These are inherently not type safe in Scala, since exceptions are unchecked), the usage of primitive data types where specific values are expected, such as passing strings to functions without wrapping or tagging them in more specific types, and unmarshalling data from type-erased sources. If we do not know whether we should expect a value to be an integer or a string and we try to read it as an integer, there is a chance to throw an exception. By this definition, accessing an external, imperative database from Scala is not type-safe. The database will have some collection of error modes that often manifest as exceptions and typically stores data in a type-erased form that needs to be unmarshalled. Finally, queries to external database systems are typically constructed as a primitive string that cannot be checked automatically at compile-time. This project addresses these issues by using carefully constructed result container types

to correctly handle error cases, including arbitrary exceptions using a pattern-match-able type hierarchy, a DSL that is type checked at compile-time by the Scala type checker, and safe, typeclass based marshalling.

2.3 Existing Graph Databases

This section gives a brief introduction into the main characteristics and classifications of existing graph databases.

2.3.1 Classes of Database

Existing graph databases typically fall into two classes: property graphs and edge-labelled (EL) graphs. https://arxiv.org/pdf/1610.06264.pdf . EL graphs typically store only a label on each edge, whereas property graphs can store more attributes on edges and are hence closer semantically to relational databases. EL graphs can be more succinctly modelled mathematically, since they can be represented purely using mathematical relations; they also lead to a cleaner syntax (see the DSL syntax subsection of the query language section). Finally, any data expressible using a property graph can be represented using an EL graph by introducing additional nodes to hold attributes.

Insert and acknowledge edge-labelled vs property graph examples from GDrive

2.3.2 Schema

Current graph database systems often do not have rigid schema. https://arxiv.org/pdf/1602.00503.pdf, introduction Instead, they use dynamic schema. This provides more flexibility in the shape of data that can be stored in the database; it also allows the types of nodes stored to evolve over time. The downside of this is that it requires additional typing checks at run-time. This also fails to fit into our definition of type safety, since we have no guarantees about any objects extracted from the database. I have instead opted for a rigid schema which is built using Scala objects and is checked by the Scala compiler at compile-time.

2.3.3 Mutability

Current graph databases tend to be mutable. In order to express concepts that require immutability, such as time-dependent data, the user has impose extra constraints, such as adding time-stamps to the schema of nodes and relations. A lack of immutability also complicates concurrent semantics and implementation of ACID transactions. I decided to introduce immutable data structures to the database, which yields ACID properties effectively for free. This is explained in section 2.4.

2.3.4 Query Languages

Typically, database systems have their own query language, the most ubiquitous of which is Neo4j's language, Cypher. This provides ML-style pattern matching of queries against

the stored graph, allowing the user to extract arbitrary fields from nodes. In most cases, theses are used by generating a query string and submitting it to the database engine. This is inherently not type-safe. The job of ensuring that absolutely every query a given program could generate is valid is intractable to undecidable for non-trivial programs. I decided to embed the query language in the host language. This allows the query language's type system to be checked by the host compiler, so we can get much stronger guarantees about program correctness. links to cypher and gremlin

2.4 Immutability

In order to implement immutability, I needed a way of creating an immutable snapshot of the database at a given point in time. This is done using a system of views. A View is such an immutable snapshot. When we read from the database, we read from a particular given view. When we write to a particular view of the database, we copy the view, update it, and return the new view. Reads and writes now never interfere with each other.

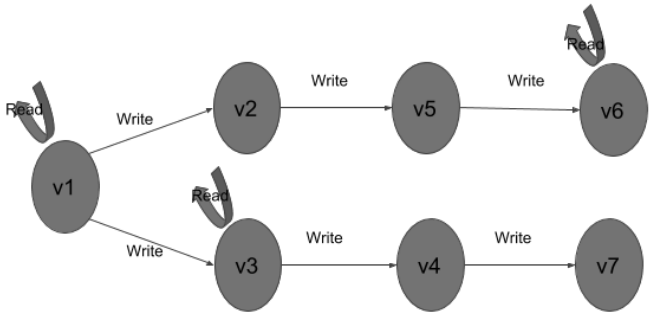

Figure 2.1: Placeholder image till I get a nicer one an example of reads and writes to an initial view.

2.5 Query Language

In this next section, I shall define the query language that I constructed and introduce a set of semantics for evaluating queries.

2.5.1 Domain Specific Language Syntax

I spent some time iterating over a DSL syntax and how I wanted queries to look and be structured. My goals were to have a highly composable and expressive, yet small language. At first I experimented with a Neo4j-style language, with pattern matching syntax, an example of which is below. The query takes a number of fields to fill in, in this case a pair of actors, and finds all valid ways to fill them in subject to the graph constraints.

```
val coactor = RelationQuery {
   (x: Pattern[Actor], y: Pattern[Actor]) => {
      val m = ?[Movie]
       (x ~ ActsIn ~> m) && (y ~ ActsIn ~> m)
   }
}
```

Find two actors such they both act in the same movie

This is fairly cluttered syntax and difficult to read. Further more, the Scala type inference system is fairly limited, meaning that users would frequently be required to insert type annotations in this syntax. It would also have been further complicated by adding the property graph features of Neo4j. Furthermore, we would not be able to easily and safely replicate Neo4j's ability to extract an arbitrary number of objects of different types from along a path defined by a query. Doing so would require use of structures such as heterogeneously typed lists.

As a result, I settled upon on an edge-labelled-relation-oriented approach which neatens the above query significantly.

```
val coactor = ActsIn --><-- ActsIn
```

Find two actors such they both act in the same movie

This is much more readable and concise. With this syntax, and the assumption that ActsIn relates Actors to Movies, Scala's typesystem can infer that coactor relates Actors to other Actors. There is also syntax for repetitions, intersections and unions. Examples in the appendix

2.5.2 Semantic Definitions

To correctly implement the DSL, we need to mathematically model its operation. Hence, the above DSL queries should correspond to an intermediate representation. I have picked a set of expression constructors to do this. These fall into two kinds: FindPair and FindSingle, indicating whether they return pairs or individual values.

FindPair queries

$$
\begin{aligned}
P \rightarrow\ & Rel(R) \text{ Find pairs related by the named relation R} \\
& \mid RevRel(R) \text{ Find pairs related by the named relation R in the reverse direction} \\
& \mid Chain(P, P) \text{ Find pairs related by the first subquery followed by the second} \\
& \mid And(P, P) \text{ Find pairs related by both of the sub-queries} \\
& \mid AndRight(P, S) \text{ Find pairs related by P where the right value is a result of s} \\
& \mid AndLeft(P, S) \text{ Find pairs related by P where the left value is a result of s} \\
& \mid Or(P, P) \text{ Find pairs related by either of the sub-queries} \\
& \mid Distinct(P) \text{ Find pairs related by P that are not symmetrical} \\
& \mid Id_A \text{ Identity relation} \\
& \mid Exactly(n, P) \text{ Find pairs related by n repetitions of P} \\
& \mid Upto(n, P) \text{ Find pairs related by upto n repetitions of P} \\
& \mid FixedPoint(P) \text{ Find the transitive closure of P}
\end{aligned}
\tag{2.1}
$$

where n denotes a natural number. These queries look up a set of pairs of objects.

FindSingle queries

$$
\begin{aligned}
S \rightarrow\ & Find(F) \text{ Find values that match the findable F} \\
& \mid From(S, P) \text{ Find values that are reachable from results of S via P} \\
& \mid AndS(S, S) \text{ Find values that are results of both subqueries} \\
& \mid OrS(S, S) \text{ Find values that are results of either subquery}
\end{aligned}
\tag{2.2}
$$

Which look up sets of single objects.

Other Definitions In order to model the operation of queries correctly, we need to define the environment in which they operate.

Object Types are the "real world" types stored in the database. These correspond to the user's Scala classes.

$$\tau \rightarrow A \mid B \mid C \mid ..$$

Named relations are the primitive relations between objects

$$R \to r_1 \mid r_2 \mid ..$$

Findables are names for defined partial functions

$$F_A \to f_1 \mid f_2 \mid ...$$

$$f\colon A \rightharpoonup \{True, False\}$$

For some given object type A. (i.e. a findable is an index into the database)

A schema, Σ, is made up of three partial functions:

$$\Sigma_{rel}\colon R \rightharpoonup \tau \times \tau$$
$$\Sigma_{findable}\colon F \rightharpoonup \tau$$
$$\Sigma_{table}\colon \tau \rightharpoonup \{True, False\}$$

Which give the types of relations and findables, and validiate the existence of a type. When it is obvious from the context, I shall simply use $\Sigma(x)$ to signify application of the appropriate function.

A view, $v \in V_\Sigma$, for a given schema Σ represents an immutable state of a database. It represents a pair of partial functions. Firstly the named-relation look up function.

$$v \in V_\Sigma \Rightarrow v_{rel}(r) \in \wp(A \times B) \text{ if } \Sigma(r) \downarrow (A, B) \tag{2.3}$$

That is, if a relation r is in the schema, then $v(r)$ is a set of pairs of objects with object type $\Sigma(r)$. Here, and from this point onwards I am using $\wp(s)$ to represent the power-set of a set, and $f(x) \downarrow y$ to mean f is defined at x and $f(x) = y$

The next function of a view is the type-look up function, it returns all objects in the view of a given object type:

$$v \in V_\Sigma \Rightarrow v_{table}(A) \in \wp(A) \quad \text{if } \Sigma(A) \downarrow True \tag{2.4}$$

That is, $v(A)$ is a set of objects of type A stored in the view, and A is a member of the schema Σ. Again I shall overload these two functions where it is clear from the context which is to be used.

2.5.3 Typing

In order to ensure the correctness of a given query, there is a type system to check it feels weird. Typing rules take two forms. Firstly typing of pair queries:

$$\Sigma \vdash P\colon (A, B)$$

Which means "under the schema Σ, pair query P returns a subset of $A \times B$". The second is for single queries:

$$\Sigma \vdash S\colon A$$

Which means "under the schema Σ single query returns a subset of A"
The rules of the first kind are as follows

$$(\text{Rel})\frac{\Sigma(r) \downarrow (A, B)}{\Sigma \vdash Rel(r)\colon (A, B)} \tag{2.5}$$

$$(\text{Rev})\frac{\Sigma(r) \downarrow (B, A)}{\Sigma \vdash Rel(r)\colon (A, B)} \tag{2.6}$$

$$(\text{Id})\frac{\Sigma(A) \downarrow True}{\Sigma \vdash Id_A\colon (A, A)} \tag{2.7}$$

$$(\text{Chain})\frac{\Sigma \vdash P\colon (A, B) \qquad \Sigma \vdash Q\colon (B, C)}{\Sigma \vdash Chain(P, Q)\colon (A, C)} \tag{2.8}$$

$$(\text{And})\frac{\Sigma \vdash P\colon (A, B) \qquad \Sigma \vdash Q\colon (A, B)}{\Sigma \vdash And(P, Q)\colon (A, B)} \tag{2.9}$$

$$(\text{Or})\frac{\Sigma \vdash P\colon (A, B) \qquad \Sigma \vdash Q\colon (A, B)}{\Sigma \vdash Or(P, Q)\colon (A, B)} \tag{2.10}$$

$$(\text{Distinct})\frac{\Sigma \vdash P\colon (A, B)}{\Sigma \vdash Distinct(P)\colon (A, B)} \tag{2.11}$$

$$(\text{AndLeft})\frac{\Sigma \vdash P\colon (A, B) \qquad \Sigma \vdash S\colon (A)}{\Sigma \vdash AndLeft(P, S)\colon (A, B)} \tag{2.12}$$

$$(\text{AndRight})\frac{\Sigma \vdash P\colon (A, B) \qquad \Sigma \vdash S\colon B}{\Sigma \vdash \Sigma \vdash AndRight(P, S)\colon (A, B)} \tag{2.13}$$

$$(\text{Exactly})\frac{\Sigma \vdash P\colon (A, A)}{\Sigma \vdash Exactly(n, P)\colon (A, A)} \tag{2.14}$$

$$(\text{Upto})\frac{\Sigma \vdash P\colon (A, A)}{\Sigma \vdash Upto(n, P)\colon (A, A)} \tag{2.15}$$

$$(\text{FixedPoint})\frac{\Sigma \vdash P\colon (A, A)}{\Sigma \vdash FixedPoint(P)\colon (A, A)} \tag{2.16}$$

$$\tag{2.17}$$

The rules for types of Single queries are similar:

$$\text{(Find)}\frac{\Sigma(f)\downarrow(A)}{\Sigma\vdash Find(f)\colon A} \tag{2.18}$$

$$\text{(From)}\frac{\Sigma\vdash P\colon(A,B)\qquad\Sigma\vdash S\colon A}{\Sigma\vdash From(S,P)\colon B} \tag{2.19}$$

$$\text{(AndS)}\frac{\Sigma\vdash S\colon A\qquad\Sigma\vdash S'\colon A}{\Sigma\vdash AndS(S,S')\colon A} \tag{2.20}$$

$$\text{(OrS)}\frac{\Sigma\vdash S\colon A\qquad\Sigma\vdash S'\colon A}{\Sigma\vdash OrS(S,S')\colon A} \tag{2.21}$$

$$\tag{2.22}$$

2.5.4 Semantics

To fully define the query language, we need to define the semantics. I have defined two collections of semantics: operational style and denotational style.

Operational Semantics Now we shall define a set of rules for determining if a pair of objects is a valid result of a query. We're interested in forming a relation $a \lhd_A Q$ to mean"a is a valid result of query Q with type A". This is dependent on the current view $v : View_\Sigma$. Hence we define $(a,b)\lhd_{(A,B),v} P$ for pair queries P and $a\lhd_{A,v} S$ for single queries S.

$$\text{(Rel)}\frac{(a,b)\in v(r)}{(a,b)\lhd_{(A,B),v} Rel(r)} \tag{2.23}$$

$$\text{(Rev)}\frac{(b,a)\in v(r)}{(a,b)\lhd_{(A,B),v} RevRel(r)} \tag{2.24}$$

$$\text{(Id)}\frac{a\in v(A)}{(a,a)\lhd_{(A,A),v} Id_A} \tag{2.25}$$

$$\text{(Distinct)}\frac{(a,b)\lhd_{(A,B),v} P\qquad a\neq b}{(a,b)\lhd_{(A,B),v} Distinct(P)} \tag{2.26}$$

$$\text{(And)}\frac{(a,b)\lhd_{(A,B),v} P\qquad(a,b)\lhd_{(A,B),v} Q}{(a,b)\lhd_{(A,B),v} And(P,Q)} \tag{2.27}$$

$$\text{(Or1)}\frac{(a,b)\lhd_{(A,B),v} P}{(a,b)\lhd_{(A,B),v} Or(P,Q)} \tag{2.28}$$

$$(\text{Or2})\frac{(a,b) \lhd_{(A,B),v} Q}{(a,b) \lhd_{(A,B),v} Or(P,Q)} \tag{2.29}$$

$$(\text{Chain})\frac{(a,b) \lhd_{(A,B),v} P \qquad (b,c) \lhd_{(B,C),v} Q}{(a,c) \lhd_{(A,C),v} Chain(P,Q)} \tag{2.30}$$

$$(\text{AndLeft})\frac{(a,b) \lhd_{(A,B),v} P \qquad a \lhd_{A,v} S}{(a,b) \lhd_{(A,B),v} AndLeft(P,S)} \tag{2.31}$$

$$(\text{AndRight})\frac{(a,b) \lhd_{(A,B),v} P \qquad b \lhd_{B,v} S}{(a,b) \lhd_{(A,B),v} AndRight(P,S)} \tag{2.32}$$

$$(\text{Exactly}\dot{}0)\frac{(a,b) \lhd_{(A,A),v} Id_A}{(a,b) \lhd_{(A,A),v} Exactly(0,P)} \tag{2.33}$$

$$(\text{Exactly}\dot{}\text{n+1})\frac{(a,b) \lhd_{(A,A),v} P \qquad (b,c) \lhd_{(A,A),v} Exactly(n,P)}{(a,c) \lhd_{(A,A),v} Exactly(n+1,P)} \tag{2.34}$$

$$(\text{Upto}\dot{}0)\frac{(a,b) \lhd_{(A,A),v} Id_A}{(a,b) \lhd_{(A,A),v} Upto(0,P)} \tag{2.35}$$

$$(\text{Upto}\dot{}\text{n})\frac{(a,b) \lhd_{(A,A),v} Upto(n,P)}{(a,b) \lhd_{(A,A),v} Upto(n+1,P)} \tag{2.36}$$

$$(\text{Upto}\dot{}\text{n+1})\frac{(a,b) \lhd_{(A,A),v} P \qquad (b,c) \lhd_{(A,A),v} Upto(n,P)}{(a,c) \lhd_{(A,A),v} Upto(n+1,P)} \tag{2.37}$$

$$(\text{fix1})\frac{(a,b) \lhd_{(A,A),v} Id_A}{(a,b) \lhd_{(A,A),v} FixedPoint(P)} \tag{2.38}$$

$$(\text{fix2})\frac{(a,b) \lhd_{(A,A),v} P \qquad (b,c) \lhd_{(A,A),v} FixedPoint(P)}{(a,b) \lhd_{(A,A),v} FixedPoint(P)} \tag{2.39}$$

$$\tag{2.40}$$

And the FindSingle rules

$$(\text{Find})\frac{a \in v(A) \qquad f(a) \downarrow True}{a \lhd_{A,v} Find(f)} \tag{2.41}$$

$$(\text{From})\frac{a \lhd_{A,v} S \qquad (a,b) \lhd_{)(A,B),v} P}{b \lhd_{A,v} From(S,P)} \tag{2.42}$$

$$(\text{AndS})\frac{a \lhd_{A,v} S \qquad a \lhd_{A,v} S'}{a \lhd_{A,v} And(S, S')} \tag{2.43}$$

$$(\text{OrS1})\frac{a \lhd_{A,v} S}{a \lhd_{A,v} Or(S, S')} \tag{2.44}$$

$$(\text{OrS1})\frac{a \lhd_{A,v} S'}{a \lhd_{A,v} Or(S, S')} \tag{2.45}$$

$$\tag{2.46}$$

Denotational Semantics The operational semantics clearly demonstrate membership of a query, but do not give a means to efficiently generate the results of query. To this end, we introduce denotations $[\![P]\!]$ and $[\![S]\!]$ such that

$$\Sigma \vdash P\colon (A, B) \Rightarrow [\![P]\!]\colon View_\Sigma \to \wp(A \times B)$$

and

$$\Sigma \vdash S\colon A \Rightarrow [\![S]\!]\colon View_\Sigma \to \wp(A)$$

Such denotations should be compositional and syntax directed, whilst still corresponding to the operational semantics.

$$[\![Rel(r)]\!](v) = v(r) \tag{2.47}$$

$$[\![RevRel(r)]\!](v) = swap(v(r)) \tag{2.48}$$

$$[\![Id_A]\!](v) = dup(v(A)) \tag{2.49}$$

$$[\![Chain(P, Q)]\!](v) = join([\![P]\!](v), [\![Q]\!](v)) \tag{2.50}$$

$$[\![And(P, Q)]\!](v) = [\![P]\!](v) \cap [\![Q]\!](v) \tag{2.51}$$

$$[\![Or(P, Q)]\!](v) = [\![P]\!](v) \cup [\![Q]\!](v) \tag{2.52}$$

$$[\![AndLeft(P, S)]\!](v) = filterLeft([\![P]\!](v), [\![S]\!](v)) \tag{2.53}$$

$$[\![AndRight(P, S)]\!](v) = filterRight([\![P]\!](v), [\![S]\!](v)) \tag{2.54}$$

$$[\![Distinct(P)]\!](v) = distinct([\![P]\!](v)) \tag{2.55}$$

$$[\![Exactly(n, P)]\!](v) = (\lambda pairs.join([\![P]\!](v), pairs))^n [\![Id_A]\!](v) \tag{2.56}$$

$$[\![Upto(n, P)]\!](v) = (\lambda pairs.join([\![P]\!](v), pairs) \cup pairs)^n [\![Id_A]\!](v) \tag{2.57}$$

$$[\![FixedPoint(P)]\!](v) = fix(\lambda pairs.join([\![P]\!](v), pairs) \cup pairs) \text{ in the domain } closure(A, v) \tag{2.58}$$

$$\tag{2.59}$$

And similarly with single queries

$$[\![Find(f)]\!](v) = \{a \in v(A) \mid f(a) \downarrow True\} \text{ for } \Sigma(f) = A \tag{2.60}$$

$$[\![From(S, P)]\!](v) = \{b \mid (a, b) \in [\![P]\!](v) \& \wedge a \in [\![S]\!](v)\} \tag{2.61}$$

$$[\![AndS(S, S')]\!](v) = [\![S]\!](v) \cap [\![S']\!](v) \tag{2.62}$$

$$[\![OrS(S, S')]\!](v) = [\![S]\!](v) \cup [\![S']\!](v) \tag{2.63}$$

$$\tag{2.64}$$

with the following definitions:

$$swap(s) = \{(b, a) \mid (a, b) \imath s\}$$

$$dup(s) = \{(a, a) \mid a \in s\}$$

$$join(p, q) = \{(a, c) \mid \exists b.(a, b) \in p \wedge (b, c) \in q\}$$

$$distinct(s) = \{(a, b) \in s \mid a \neq b\}$$

$$filterLeft(p, s) = \{(a, b) \in p \mid a \in s\}$$

$$filterRight(p, s) = \{(a, b) \in p \mid b \in s\}$$

I have also proved in the appendix the correspondence of these two formulations of semantics.

2.5.5 Commands

In addition to the query construction language, there are five commands which make use of the queries.

format this correctly

$$findPairs(P, v) = [\![P]\!](v)$$
$$find(S, v) = [\![S]\!](v)$$
$$shortestPath(a, a', P, v) = \text{Shortest path from } a \text{ to } a' \text{ in the graph defined by } [\![P]\!](v)$$
$$allShortestPaths(a, P, v) = \text{The shortest path to each element reachable from } a \text{ in the graph defined b}$$

The definition of the write command is slightly more complicated. It needs to return an updated view with its look up functions now returning the inserted objects. Write takes a collection, rs, of correctly typed named relations instances.

$$rs = \{(a_i, r_i, b_i) \in (A \times R \times B) \mid 0 < i \leqslant n\} \text{ for some } n \text{ being the size of the set.}$$

and updates the view as so:

$$\begin{aligned} write(rs, v) = & v\,[A \mapsto v_{table}(A) \cup \{a_i \mid 0 < i \leqslant n\}] \\ & [B \mapsto v_{table}(B) \cup \{b_i \mid 0 < i \leqslant n\}] \\ & [r \mapsto v_{rel}(R) \cup \{(a_i, b_i) \mid 0 < i \leqslant n\}] \end{aligned} \tag{2.65}$$

2.5.6 Summary

To summarise, this past section defines a query DSL, a type system for checking validity of queries, and a set of semantics for executing queries. Furthermore, I have then defined a set of commands which allow queries to executed and views to be generated.

2.6 Languages and Libraries Used

In addition to Scala, I have made use of several libraries and tools. Some were used for the actual construction of the project, whereas others were used for auxiliary tasks such as testing and debugging or generating datasets.

SBT Scala Build Tool (SBT) is a package manager and build tool for Scala which allows access to Java libraries. https://www.Scala-sbt.org/

PostgreSQL An open-source, multi-platform SQL implementation. https://www.postgresql.org/

LMDB An open source, highly optimised key-value datastore. https://symas.com/lmdb/

Scalaz A library for Scala providing typeclasses and syntax to aid advanced functional programming. https://github.com/Scalaz/Scalaz

JDBC Java's standard library support for SQL connections. Used to interact with a postgres database. http://www.oracle.com/technetwork/java/javase/jdbc/index.html

LMDBJava A JNI library allowing access to the LMDB datastore. Previously, I experimented with another library, LMDBJni. However, upon discovery of a bug in the JNI code, I switched to LMDBJava. https://github.com/lmdbjava/lmdbjava

Junit A unit testing library for JVM languages. https://junit.org/junit5/

SLF4J A logging library for JVM languages https://www.slf4j.org/

Spray-JSON A JSON library for Scala, used to generate benchmarking datasets. https://github.com/spray/spray-json

Python A scripting language used to generate test datasets. https://www.python.org/

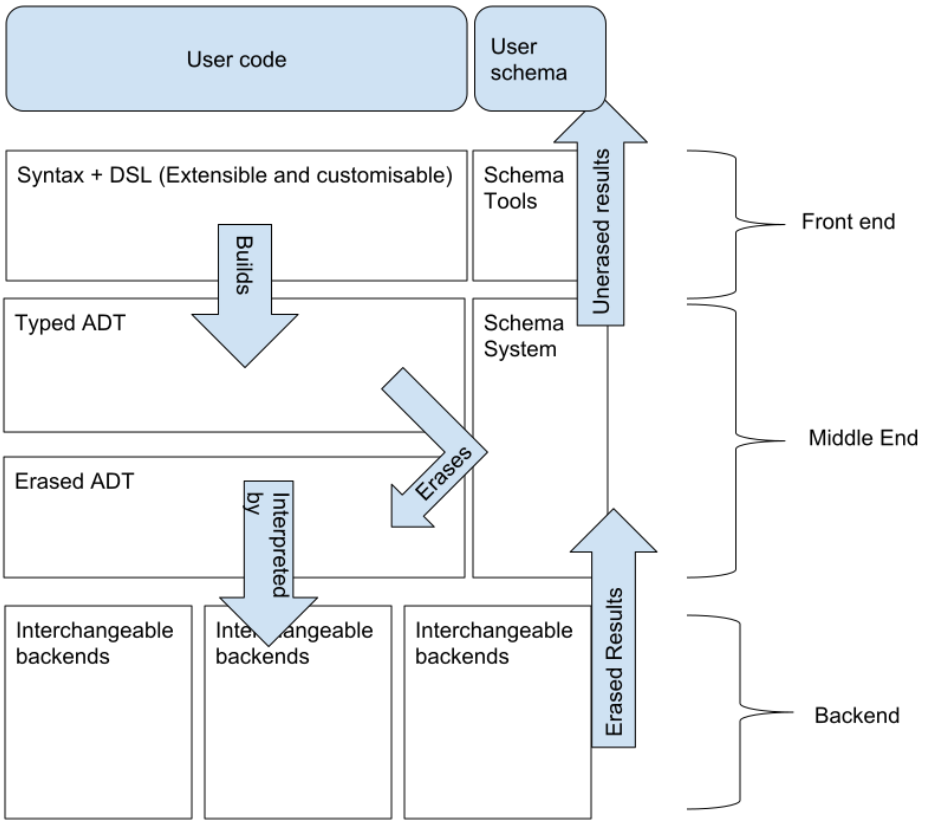

Figure 2.2: A map of the architecture of the system

2.7 Software Engineering

The system was designed to be as modular and re-implementable as possible. To this end, it consists of several interchangeable components; a map of this is provided in figure 2.2. The front-end mostly consists of interfaces and syntax for building queries and the schema for the database. The front-end's DSL translates into the underlying typed ADT. This then has its compile-time type information erased to become an un-typed query ADT, which is easier and cleaner to interpret. Finally, there are three interfaces that a back-end system must implement: DBBackend, DBInstance and DBExecutor. These specify a root back-end object that opens an instance which represents a database connection. Each DBInstance has an executor that allows us to execute commands. The results of these commands are then un-erased to return them as the correct type.

2.8 Scala Techniques

I have made use of several advanced programming techniques which are specific to the Scala language.

2.8.1 Type Enrichment

The type enrichment feature of the Scala language allows retroactive addition of methods to previously defined types. Type enrichment is performed by creating an implicit class, taking a single underlying value of some given type. The methods defined in the implicit class can now be called as if they were methods of the underlying class, provided the implicit class is in scope.

```
object Outer {
    implicit class EnrichInt(i: Int){
        def neg: Int = -i
        def  succ: Int = i + 1
        def mod(n: Int) = i - (i / n)*n
    }
}
```

When Outer is imported, we can now do

```
5.neg == -5
6.succ == 7
10 mod 4 == 2
```

This allows us to add syntax to types where a small number of core methods have been defined.

2.8.2 Implicit Parameters

Another advanced feature of Scala that I have used is that of implicit parameters to functions and class constructors. Values such as vals, functions, and classes can be declared with an implicit tag. Functions and classes can declare additional parameters as implicit. When these functions are called, the implicit parameter can be omitted if and only if there is an unambiguous implicit value of the correct type in scope.

```
object foo {
    implicit val i = 5
}

object bar {
    def plusImpl(j: Int)(implicit k: Int): Int = j + k
}
```

```
import foo._

bar.plusImpl(4) == 9
```

This is typically used for purposes such as passing around values that are defined once and used many times in a program, such as an execution context, or a logging framework instance.

Functions can also be implicit and take implicit values

```
implicit def f(implicit a: A): B = ...
```

When an implicit value of type A is available, then an implicit value of type B is also available.

This can be made more interesting when we include parameterised generic types. This allows us to get the compiler to do work in the manner of an automated theorem prover (by the curry-howard correspondence) at compile-time.

```
object Defs {
    implicit val s: String = "foo"
    implicit val i: Int = 2
    Implicit def toList[A](implicit a: A): List[A] = List(a, a, a)

    def getImplicit[A](implicit a: A): A = a
}
```

```
import Defs._

getImplicit[List[Int]] == List(2, 2, 2)
getImplicit[List[List[String]] ==
        List(List(foo", foo", "foo"),
        List("foo", "foo", "foo"),
        List("foo", "foo", "foo"))
```

2.8.3 Typeclass Pattern

A further combination of these two patterns is the typeclass pattern. We define a typeclass for a type by defining as methods on a trait the operations we want on the type. We can then define implicit objects which work as type class instances for the types we want. We can also use implicit functions to generate typeclass instances in a manner similar to a proof tree. We can then use the type enrichment feature to add methods to values of a type that is a member of the typeclass.

```
trait Monoid[A] {
     def id: A
     def op(a1: A, a2: A): A
}
```

Definition of a monoid typeclass instance using F-type polymorphism

```
object Instances {
     implicit object IntMonoid extends Monoid[Int] {
        override val id: Int = 0
        override def op(a1: Int, a2: Int): Int = a1 + a2
     }

    implicit def ListMonoid[A] = new Monoid[List[A]] {
            override val id: List[A] = List()
            override def op(a1: List[A], a2:  List[A]):  List[A= a1 ++ a2
      }


    implicit def PairMonoid[A, B](implicit MA: Monoid[A], MB: Monoid[B]) =
     new Monoid[(A, B)]{
      override val id: (A, B) = (MA.id, MB.id)
      override def op(p: (A, B), q: (A, B)) =
       (MA.op(p._1, q._1), MB.op(p._2, q._2))
    }
}
```

Definition of several monoid instances, including a pair monoid that combines monoids.

```
object Syntax {
        implicit class MonoidOps[A](a: A)(implicit ma: Monoid[A]) {
          def *(n: Int): A = if (n == 0) ma.id else ma.op(a, *(n-1))
        }
}
```

Definition of a multiplication operation on instances of a monoid.

```
(5, List("f")) * 3 == (15, List("f", "f", "f"))
```

Usage of the multiplication operation on a tuple.

It is clear that these patterns allow for very expressive structures and abstractions to be built in Scala. I use these frequently within the project to neaten code and to achieve type-safety.

Chapter 3

Implementation

In this section, I shall first focus on common techniques when building the database, then explain construction of the front- and middle-ends. After this, I shall look at common structures shared by the various backend implementations and finally, I shall conclude with an examination of each of the backend implementations that I have written.

3.1 Note on Purity and Concurrency

All of the back-ends aim to preserve the global immutability of the database. The immutable semantics of the database also mean that queries generally do not interfere with each other. This means that we can avoid keeping locks or creating large transactions. Hence, the backends do not require much work to maintain correct concurrency.

3.2 Functional Programming Techniques

3.2.1 Monadic Compilation

At several points in building a backend, it becomes necessary to transform one algebraic type to another. This is typically done by walking over a tree, whilst keeping some mutable state representing parts of the tree that are relevant. One example is converting an intermediate representation to SQL output code. Here, we may want to extract common subexpressions into a dictionary, or pick out all the database tables that need to be accessed by the query. This can be encoded by folding a State Monad instance over the tree.

The state monad is an abstraction over functions that chain an immutable state through successive computations.

```
class State[S, A](r: S => (S, A)) {
      def runState(s: S): (S, A) = r(s)
}


object Example {
    Implicit def StateMonadInstance[S] = new Monad[State[S, _]] {
         def point[A](a: A) = new State(s => (s, a))
```

```
        def bind[A, B](sa: State[S, A], f: A => State[S, B]) = new State(
           s => {
                 val (s', a) = sa.runState(s)
                 f(a).runState(s')
             }

        )
    }
}
```

Do I want to link to this instead?

To define a monadic compiler, we define a recursive function which chains state monad objects together.

```
def compile(adt: ADT): State[S, Res] = adt match {
   case basis => ...
   case Pattern(a, b) => for {
      ca <- compile(a)
      cb <- compile(b)
   } yield foo(ca, cb)
}
```

3.2.2 Constrained Future Monad

Part of the goal of type-safety is the recovery of error cases. Typically, this would be done in a JVM program through the use of exceptions. However the presence of unchecked exceptions on the Java platform makes it difficult to ensure that all error cases are accounted for. A more functional approach is the use of the exception monad. http://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf In Scala, this manifests as the built in Try monad and Scalaz's \/ (Either) monad. Try[A] has two case classes: Failure(e: Throwable) and Success(a: A), while E \/A has the case classes -\/(e: E) and \/-(a: A) (Left and Right). Since Try's failure case is the unsealed Throwable trait, we do not really have a way to ensure we have handled all error cases at compile-time, whereas Either has a parametrised error type, which can be a sealed type hierarchy, which is then checked by the Scala compiler at run-time. For example, consider the simple interpreter below. All error cases are proved to be handled by the type-system.

```
sealed trait Error
case object DivByZero extends Error
case object Underflow extends Error

sealed trait NoError
```

A simple pair of error hierarchies. One signifying errors, and NoError indicating a lack of error. Note that *NoError* cannot be instantiated or sub-classed.

```
def eval(e: Expr): Error \/ Int =
    e match {
         case Div(a, b) => for {
               aRes <- eval(a)
               bRes <- eval(b)
               r <- if (bRes == 0) DivByZero.left else (aRes/bRes).right
            } yield r

        case Sub(a, b) => for {
            aRes <- eval(a)
            bRes <- eval(b)
            r <- if(bRes > aRes) Underflow.left else (aRes-bRes).right
    } yield r
}
```

A simple interpreter for a simple algebraic datatype for expressions that returns an error or a result

```
def evalAndPrint(e: Expr): NoError \/ String =
    eval(e) match {
        case \/-(i) =>i.toString.right
        case -\/(e) => e match {
                 case DivByZero => division by zero.right
                 case UnderFlow => underflow.right
}}
```

A simple result printer that correctly handles all error cases and reduces its error parameter to the un-instantiable *NoError* type

Typically, we're dealing with cases which might take a significant amount of time to return. So rather than using a simple Either monad, we lift it into an asynchrony monad. There are several options to choose from for an asynchrony monad. I chose the built in Future over more exotic alternatives, such as the Scalaz Task, since it is relatively widely used, and I have some familiarity with it from past projects. Futures also capture thrown unchecked exceptions, which makes handling them a little easier. Hence we're interested in passing around Future[E \/A] around, for a sealed type hierarchy E. There is also the issue of the java libraries used (for SQL and LMDB access) throwing exceptions, and unexpected exceptions turning up in code. Fortunately, the Future container catches these, acting like an asynchronous Try[E \/A]. This causes issues as we do not have the sealed trait property of errors as we have above. To solve this, I introduced the ConstrainedFuture[E, A] monad, which has the requirement that the error case type parameter E implements the HasRecovery typeclass for converting any Throwable to an E.

```
trait HasRecovery[E] {
 def recover(t: Throwable): E
}
```

The underlying future is kept private, and can only be accessed via the run method, which calls the recover method on any errors (tail recursively, so any exceptions thrown during execution are also handled). By this construction appendix we ensure all non-fatal error cases are contained in a type-safe way.

3.2.3 Operation Monad

As stated, Can I link to it? typical database operations take a view as a parameter, inspect the view, return a value and may also insert a new view. This requires interplay between the ConstrainedFuture (To handle failure and asynchrony) and State (to chain together updates to the view representing current state) monads. Hence we use the Operation[E, A] monad, which wraps a function $(ViewId \Rightarrow ConstrainedFuture[E, (ViewId, A)])$ in a similar way to how State monad chains together functions $S \Rightarrow (S, A)$. Each of the commands on a database yields an operation.

3.2.4 Local and Global State

Although it would be preferable to only use purely functional folds, maps, and immutable data structures everywhere within the project, for certain, high frequency, performance critical tasks, using purely immutable structures slows us down. Recursive functions (though my functional style does not make heavy use of them anyway) tend to use more stack space than the JVM has available in many situations. Hence for tasks such as retrieving values for result sets, pathfinding across large relations, and building indices, I have opted to make use of mutable data structures locally, using builders for collections such as sets. This leads to a dramatic increase in speed, especially for when large numbers of elements are added sequentially Source?. In these cases, the mutable state never leaks out of the functions that make use of the mutability.

3.3 Schema Implementation

One of the goals of the project was to allow close to arbitrary user objects (assuming that they are finite) to be inserted and retrieved from the database. This was achieved using the typeclass pattern.

3.3.1 Schema Hierarchy

In order to work with the database, a class needs to have an instance of the SchemaObject typeclass.

```
sealed trait SchemaObject[A] {
// components for constructing DB
 def schemaComponents: Vector[SchemaComponent]  tables
 def any: Pattern[A] // Findable that matches any A
 def findable(a: A): DBTuple[A] // convert an object to a findable
 def name: TableName // name of the table
 def fromRow(row: DBObject): ExtractError \/ A // unmarshall an A
}
```

SchemaObject is sealed, so can only be implemented by implementing one of its subclasses. Currently, for the sake of simplicity, there are five: SchemaObject0 .. SchemaObject4, with the number indicating the number of underlying database fields required. These simplify the construction of a type-class instance to marshalling values of the object type to tuples of Storable" (read: primitive) types.

To go in the appendix: The definition of a schema object subclass

3.3.2 DBObjects

To store objects in the database under various backends, we need to have a type erased (at compile-time, but not run-time) version of the objects. Once we have a primitive representation of an object from the SchemaObject, we can fully unerase it by converting it to a DBObject. A DBObject is simply a collection of type tagged fields (DBCell). These can now be easily inserted or retrieved from a database.

```
type DBObject = Vector[DBCell]

type DBCell -> DBInt of Int
|  DBString of String
| DBBool of Boolean
| DBDouble of Double
```

Pseudo-ml definitions of DBObject and its components

3.3.3 Unerasure

In order to correctly retrieve values from the database, we need to be able to undo the erasure process. This can be done using the unmarshalling methods derived from the SchemaObject for an object type.

3.3.4 Relations

In order for type checking to work, relations need to have type parameters for the object types they link. Hence, to define relations in the schema, the user needs to define objects that extend the relation interface.

```
abstract class Relation[A, B](implicit val sa: SchemaObject[A], val sb: SchemaObject[B])
   extends FindPairAble[A, B]
```

3.3.5 SchemaDescription

In order to build the database structures the backends need a definitive collection of schema to include. This is performed using a SchemaDescription object, which simply holds a collection of SchemaObjects and Relations which need to be used by the database.

3.3.6 Findables

For the sake of simplicity, I have only implemented findables which test if fields of objects match particular values. This allows us to look for specific objects or match particular fields. This also makes indexing easier to implement.

3.4 Query ADT

The intermediate representation terms described in the preparation section link are implemented in Scala by a pair of ADT hierachies: a typed and type-erased equivalent for each. The typed ADTs are parameterised by the object types which they look up. Using this parameterisation, I have encoded in type of the ADTs the inductive type rules in the semantics such that the Scala compiler checks the type of any generated query and does type inference for us. Link to Appendix "The definitions of these types can be found in the appendix..." The only rules that cannot be checked at compile-time are whether the schema description contains the relation in instances of the (Rel) rule, the type A for the (Id_A) rule, and the (Find) rule, as we cannot predict the contents of the SchemaDescription at compile-time without dependent types. The typed ADTs are erased, with respect to a schema, into their unsafe equivalents in order to be executed. If an AST node makes reference to a non-existent table or relation, a run-time error is created in the Either return type. The constructors of the type-erased ADT nodes are private to the enclosing package, meaning that they can only be created by erasing a typed ADT.

3.5 Commands

As specified in the previous chapter, each backend needs to implement five commands:

- find(S): Operation[E, Set[A]]
- findPairs(P): Operation[E, Set[(A, B)]]
- ShortestPath(start, end, P): Operation[E, Option[Path[A]]
- allShortestPaths(start, P): Operation[E, Set[Path[A]]
- insert(relations): Operation[E, Unit].

Each command should return an Operation of the correct type.

3.6 DSL

The DSL mostly consists of syntactic sugar to make queries easier to read. It is implemented using the type enrichment pattern. Both Relation and FindPair implement the trait (interface) FindSingleAble, so we can use type enrichment to write new DSL opertors. The main thing to note in the DSL is the use of arrows to chain relations together. (See RelationSyntax.Scala in the appendix for examples of DSL). Put in appendix

3.7 Common Generic Algorithms

During construction of the database, several common patterns of problems emerged with slight differences. Hence, I have written relatively optimised generic versions of these algorithms such that different backends can make use of them regardless of the underlying types. These algorithms are found in core.utils.algorithms

3.7.1 Simple Traversal

The first set of generic algorithms to look at are the SimpleFixedPointTraversal algorithms. These compute the repetitions of a function for execution of the FixedPoint, Upto, and Exactly expressions of the ADT. They are labelled as "Simple" because they do the search from a single root. They carry out search mutably, and convert their output to an immutable set upon returning.

Exactly The simplest algorithm is for computing Exactly query nodes. Here, we simply expand a fringe set of values outwards, by applying the search step to every value in the fringe to get the new fringe. We also memoise the search step function in the case of repetitions. After the required number of repetitions, the remaining fringe is returned. (Diagram showing expanding fringe)

Upto The next algorithm is for computing Upto. This is computed in a similar way by flat-mapping the functions over the fringe repeatedly to calculate an expanding fringe. The major differences here are that we keep an accumulator of all the found values. When a new fringe is calculated, we subtract the accumulator set from it to reduce the number of nodes that need to be searched to those that have not yet been searched, and then union the remaining fringe with the accumulator to get the new accumulator. After the required number of repetitions, the accumulator is returned. diagram

FixedPoint The final algorithm is to calculate FixedPoint. This works slightly differently. As before, we keep an accumulator of reached nodes, but unlike before, the generation number of each node is not important, only that a node is reachable is important. Hence, the fringe is a queue rather than a set, and we iterate until the fringe is empty. In each iteration, we pop off the top value of the fringe queue, compute all immediately reachable nodes. This reachable set is diffed (define diff) with the accumulator to find the newly reached nodes. These are now added to the fringe queue and the accumulator. When the fringe is empty, we return the accumulator.

3.7.2 Full traversal

The next set of algorithms build on the SimpleFixedPoint algorithms to return not just those nodes reachable from a single root, but the set of all reachable pairs with the left hand pair derived from a root set (using the left-hand optimisation). As such, the algorithms need to do some more work to reconstruct which nodes are reachable from each root, while still eliminating redundancies.

Exactly The first such algorithm is for computing Exactly. This is similar to the original version, except we now store a fringe for each root in a map of Root =¿ Set[Node]. We also memoise the search function in a Map to avoid computing it redundantly. In each iteration, we simply expand the fringe for each root as before by mapping the fringe expansion loop body over the values of the fringe map. Diagram

Upto The next algorithm is to compute Upto. This is again done like before, but with a map of root to accumulator set as the accumulator. As with Exactly, the fringes of each root are expanded simultaneously, sharing redundant results via the memo, while the accumulators are unioned with the fringe of the appropriate root with each iteration. Diagram

FixedPoint Finally, the FixedPoint take a departure from the parallel implementations above. The reachable set of each node is calculated sequentially using a similar algorithm to above. However the memo now contains all nodes reachable from previously processed roots, allowing for fast convergence of dense graphs.

3.7.3 Pathfinding

The ShortestPath and AllShortestPaths commands require us to search a subgraph generated by a relation. Since all edges have unit weight, the pathfinding algorithm reduces to breadth-first-search, which I have implemented in an imperative format while wrapping up error cases in an Either. These functions take, as a parameter, the search step A ⇒ E \/Set[A] which represents the edges going out of a node.

3.7.4 Joins

The problem of joining two sets of pairs based on shared intermediate values is a requirement for any backend.

$$A\ join\ B = \{(a, c) \mid \exists b.(a, b) \in A \wedge (b, c) \in B\}$$

I have implemented a simple hash join (http://www.csd.uoc.gr/ hy460/pdf/p63-mishra.pdf). This operates by first assuming that the size of distinct leftmost elements in the right set is smaller (known to be a subset of, due to the left-optimisation of backends) than the rightmost elements of the left set. We then build a mutable map to index leftmost values to rightmost values. We then traverse the left set and for each pair generate all new joined pairs, using a flatmap to collect the values into one set. An issue with this approach is that we have to build the index upon each join call, an issue that is addressed later.

```
def joinSet[A, B, C](
    leftRes: Set[(A, B)],
    rightRes: Set[(B, C)]
): Set[(A, C)] = {
  // build an index of all
  // values to join on right, since Proj_B(right)
  // is a subset of Proj_B(Left)
  val collectedRight = mutable.Map[B, mutable.Set[C]]()
  for ((b, c) <- rightRes) {
    val s = collectedRight.getOrElseUpdate(b, mutable.Set())
    s += c
  }

  for {
```

```
      (left, middle) <- leftRes
      right <- collectedRight.getOrElse(middle, Set[C]())
    } yield (left, right)
  }
```

3.8 Views and Commits

In the memory backend, as will be explained later, views are easy implement as a map of ViewId to MemoryTree and simply updating the MemoryTree, allowing Scala's immutable collections to handle sharing of data in an efficient way. In the other backends, backed by non-functional technologies, we need other methods of sharing and inserting to immutable structures. A first observation is that with the operation monad model, each view only has one direct predecessor, forming a tree where we're only interested in the path back to the root from a given node. From this insight, we can think of each new view only needing to store a difference against its parent. Since the current design of the database only allows the addition of values and not deletion, this difference is only going to be positive. Hence, we can store all the added values between two views in a container called a Commit. As an optimisation, we can disregard the parent view, and simply store each view as a collection of its commits. This would also allow us to implement deletion if desired. This would be done by removing commits in a view that are to be modified, and replacing them with a new commit containing the contents of those commits, excluding the deleted values.

3.9 Memory Backend

The first backend that I have implemented is a simple, naive memory-based backend. This backend follows the denotational semantics, and makes very few attempts to improve performance. This backend serves as a test-bench backend, used to create unit tests to test other backends. It also allowed me to practice implementation of type erasure and unerasure according to the schema in a controllable environment (that is, without interference from other languages or libraries as in the SQL and LMDB backends.)

3.9.1 Table Structure

A memory instance stores a concurrent map of ViewId to MemoryTree, which itself is a map of TableName (derived from the SchemaDescription) to MemoryTable. There is a MemoryTable for each object class in the SchemaDescription. A MemoryTable provides lookups using maps for DBObjects and Findables, in the form of an index to set of objects for each column value. These lookups return objects containing a DBObject and the outgoing and incoming relations for the object, indexed by RelationName. Diagram of View =¿ Tree(TableName =¿ Table(Index =¿ Object))

3.9.2 Reads

Reads occur by simply walking over the ADTs recursively, following the denotational semantics Link to appendix. The only real departures from the denotational semantics are the Left-optimisation and use of the generic fixed point algorithms to compute Exactly(n, P), Upto(n, P), and FixedPoint(P). Results are also computed in the Either monad to allow for error checking (for cases such as missing tables).

3.9.3 Left Optimisation

One of the few optimisations here is the Left optimisation. When we compute the result set of a FindPair, we pass in the subset of left hand side variables we want to compute from. This mostly has an effect when we compute joins (Chains). Consider joining a query of maybe a few dozen unique rightmost values to a large query with several million unique leftmost values. The pairs of the right relation only need to be joined if their leftmost value is in the set of rightmost values of the left relation. Hence it makes sense to pre-limit our search to pairs with leftmost

values in the right most value. (Diagram showing redundant calculations in the join). This pattern also makes an appearance in the original LMDB implementation.

3.9.4 Writes

Thanks to Scala's immutable collections library, updating the database is fairly easy When inserting a collection of relations, we simply add, relation by relation, each object in the relation, if it does not exist, and update the outgoing and incoming relation map of each object. This update creates new immutable object tables and a new memory tree. This is stored to the map of ViewId to MemoryTree as a new view.

3.9.5 Storage

Objects are stored as DBObjects in wrapper MemoryObjects. MemoryObjects are hashed and compared by their DBObjects. Memory object diagram

3.9.6 Mutability

The only mutability in the Memory implementation is for the views counter and the views lookup table. Access is kept transactional using a lock.

3.9.7 Pathfinding and fixed point traversal

The pathfinding and fixed point traversal methods are simply implemented by calls to the appropriate generic algorithms using the interpreted findPairs function as the search function.

3.10 PostgreSQL backend

Upon completion of the initial memory backend, I started work on a PostgreSQL based backend. This compiles the ADT intermediate representation into SQL queries that are then executed by a Postgres database.

3.10.1 Table Structure

The construction of the underlying database uses several SQL tables. These can be partitioned into a set of control tables that will be present in all database instances and a set of schema defined tables.

Control Tables

Name	Schema	Description
Default	(ViewId)	Holds the mutable default ViewId
Commits Registry	(CommitId^{primary}, Dummy)	Stores which commits are valid, allows us to give each commit a unique id
Views Registry	(ViewId^{primary}, Dummy)	Stores which views are valid, allows us to give each view a unique id
Views Table	(ViewId, CommitId)	Stores which commits belong to which view.

Note: ViewId is a foreign key into the views registry, CommitId is a foreign key into Commits Registry. The Dummy column is needed to fix postgresql syntax when we try to get the next key for a table with one column.

Schema defined Tables

Control Tables

Name	Schema	Description
Object Table	(ObjectId^{primary}, generated)	Generated per schema object in schema description. A schema column is generated per schema object field, with an appropriate SQL type. This allows indexing by findables to get object Ids for queries, and extraction of objects from the database.
Auxiliary Table	(ObjectId, CommitId)	One auxiliary table is associated with each object table. It stores the object Ids associated with each CommitId, which is a many-many relation
Relation	(LeftId, CommitId, RightId)	Generated per relation in schema description. Each is associated with a commit.

Note: Object-, Left-, and Right- Ids are foreign keys to the ObjectTable's ObjectId column.

3.10.2 Query Structure

n order to manage the complexity of queries, I have made use of SQLs Common Table Expressions feature. CTEs act like let expressions in ML. (https://www.researchgate.net/publication/242270488˙Common˙Table˙Expression˙-˙WITH˙Statement)

The first CTE we use selects all the valid CommitIds.

```
WITH RECURSIVE VIEW_CTE AS
(SELECT $commitId FROM $viewTable WHERE $viewId = $id)
```

The next collection of CTEs select the valid instances of each relation from the relation tables.

```
WITH Relation0 AS (SELECT left_id, right_id
FROM REL_0_0 JOIN VIEW_CTE ON REL_0_0.CommitId = VIEW_CTE.CommitId)
```

We then select the valid instances from the auxiliary tables the values we use for Id_A queries.

```
AuxTable2 AS (SELECT DISTINCT obj_id AS left_id, obj_id AS right_id FROM AUXUSERSPACE_Peop
```

We then create CTEs for any repeated subqueries (ie those used by FixedPoint, Exactly, and Upto)

```
View1 AS (SELECT * FROM Relation0),
```

We then create a CTE for the main query, which exposes a leftId and rightId column

```
main_query AS (SELECT left_id, right_id FROM View3 )
```

Finally, we join the main query CTE to queries to extract the left and rights .

```
(SELECT left_table.col_0 AS left_col_0, right_table.col_0 AS right_col_0
FROM (
(USERSPACE_People_0 AS left_table JOIN main_query ON left_table.obj_id = main_query.left_i
JOIN USERSPACE_People_0 AS right_table
ON right_id = right_table.obj_id)
)
```

The main query itself is built compositionally, in that each subquery exposes a leftId and rightId, allowing queries to be composed to form larger queries in a standard form. A full query might look like this: Colour code this query, make it fit

```
WITH RECURSIVE
VIEW_CTE AS (SELECT CommitId FROM VIEWS_TABLE WHERE ViewId = 1),
Relation0 AS (SELECT left_id, right_id
FROM REL_0_0 JOIN VIEW_CTE ON REL_0_0.CommitId = VIEW_CTE.CommitId),
AuxTable2 AS (SELECT DISTINCT obj_id AS left_id, obj_id AS right_id FROM AUXUSERSPACE_Peop
View1 AS (SELECT * FROM Relation0),
View3 AS (
(SELECT left_id, right_id FROM AuxTable2 )
UNION
SELECT View3.left_id AS left_id, View1.right_id AS right_id
FROM
View3 INNER JOIN View1 ON View3.right_id = View1.left_id
),
main_query AS (SELECT left_id, right_id FROM View3 )
(SELECT left_table.col_0 AS left_col_0, right_table.col_0 AS right_col_0
FROM (
(USERSPACE_People_0 AS left_table JOIN main_query ON left_table.obj_id = main_query.left_i
JOIN USERSPACE_People_0 AS right_table
ON right_id = right_table.obj_id)
)
```

3.10.3 Monadic Compilation

In order to generate these SQL queries, we need to convert the ADT query to a lower level intermediate representation In appendix, which maps one-to-one to SQL. While doing this, we need to gather and rename all of the relation and auxiliary tables that we extract from so that we can form the CTE queries, we also need to find repeated queries to hoist out. In order to do this, we use the monadic compiler pattern described above. The compiler state is shown below

name the figure. When the compilation is done, depending on the context of the command, we append different extraction queries. For find pairs we need to extract the fields of both the left hand side and the right hand side objects, whereas for pathfinding operations, we only need to extract the ObjectIds along the path as opposed to whole objects. format diagrams here

```
case class CompilationContext(
        varCount: Int, // Unique identifier for variable names
        relations: Map[ErasedRelationAttributes, VarName], // Relations used by the query
        requiredTables: Map[TableName, VarName], // tables needed for Findables in the que
    requiredAuxTables: Map[TableName, VarName], // the aux (commit) tables needed for the
        commonSubExpressions: Map[SubExpression, VarName]
)
```

3.10.4 Writes

There are several steps in the implementation of writes, though I have not expended a great deal of effort making them fast. A significant part of this is the "insert or get" SQL query, which looks up an object in the relevant table, returning its ObjectId if it exists and creating the object and returning the new ID if it does not.

```
WITH insertOrGetTemp AS (
  INSERT INTO $objectTable ($columnNames)
  SELECT $values WHERE NOT EXISTS (SELECT 0 FROM $objectTable WHERE $ValueConditions)
  RETURNING $objId
) SELECT * FROM insertOrGetTemp  UNION ALL (SELECT $objId FROM $name WHERE $ValueCondition
```

We create a new view and commit, then we run a memoised InsertOrGet over all the leftmost objects to be inserted, then all the right objects to be inserted, yielding tagged relations between ObjectIds. For each inserted relation, the existing relation instances are removed from those to insert, and the remaining inserted to the relevant RelationTable with the correct CommitId. The auxiliary tables are now updated and, on success, the views table is updated.

3.10.5 Mutability

As with the memory backend, all mutability except for the availability of views and the default view is hidden from the user. The SQL backend uses commits to manage view mutability.

3.10.6 Pathfinding and Fixed Point Traversal

Pathfinding is implemented by constructing an SQL query to generate right hand side ObjectIds for a relation given a left hand side ObjectId. This query is used as the search function for the generic pathfinding algorithms. Once paths have been found, their ObjectIds are looked up in the database to find the full values along each path.

Fixed point traversal and repetitions are done natively in SQL. FixedPoint and Upto are done using a recursive CTE, while Exactly is done by explicitly joining together the required number of repetitions of the sub-relation's query.

```
WITH RECURSIVE
    RecursiveCTE AS (
     (SELECT *, 0 AS Counter FROM BasisCase )
     UNION
```

```
    SELECT
    RecursiveCTE.left_id AS left_id,
    InductiveCase.right_id AS right_id,
    Counter + 1 AS Counter
    FROM RecursiveCTE INNER JOIN InductiveCase
    ON RecursiveCTE.right_id = InductiveCase.left_id
    WHERE Counter < Limit
   )
```

An example of a recursive CTE computing Upto. FixedPoint would omit the counter variable and the limit.

3.10.7 Object Storage

Objects are stored as in the appropriate ObjectTable in a manner derived from the DBObject of each object. Each DBCell is converted to appropriate SQL type.

3.11 LMDB Backends

The final family of backends are the LMDB backends. LMDB is a simple, efficient memory mapped file based Key-Value datastore. (https://symas.com/lmdb/). It is already the backend for several DBMSs (Source?).

3.11.1 Common

Although I have written several versions of the LMDB backend, there are several overarching similarities.

LMDB API

Terminology LMDB terminology differs slightly from that of more mainstream systems. What would be called a database in SQL is known as an Env and a table known as a DBI.

Transactions LMDB's transactional model differs from more traditional systems. In short, we need to create a transaction to do reads but not writes. This stems from LMDB delegating as much work as possible to the OS kernel's memory-mapped file system. All writes are immediately written to the database, whereas starting a read transaction gives a view of the Env as it was at the start of the transaction, ensuring that concurrent writes don't affect what is read from the database. Writes may be included in transactions to allow atomic get-and-sets. In practice the transactions we actually use are very short lived.

JVM API The LMDB backend uses the LMDBJava API https://github.com/lmdbjava/lmdbjava to allow access to an LMDB Database from JVM languages (such as Scala). For each DBI, we have a byte array to byte array key-value map, much like a Map[Array[Byte], Array[Byte]].

Storage and Keys In order to write to the database using the LMDBJava API, we need to convert the key and value to ByteBuffer. To generify this process I have introduced two type classes: Keyable and Storable.

```
trait Keyable[K] { def bytes(k: K): Array[Byte]}

trait Storable[A] {
 def bufferLength(a: A): Int
 final def toBuffer(a: A): ByteBuffer =
```

```
    {/*Allocates a buffer and fills it using declared methods*/}
  def writeToExistingBuffer(a: A, buf: ByteBuffer): Unit
  def fromBuffer(buf: ByteBuffer): LMDBError \/ A
}
```

Keyable is a type class which shows that an object can be converted into an array of bytes representing a component of a key (Can be concatenated with others to make a full key). Storable is a more general typeclass that shows that objects can be converted into a ByteBuffer. Storable objects are typically more complex than objects used as keys, and have a higher marshalling throughput, hence the lower level, faster, ByteBuffer API. Typeclass instances exist to allow sets and lists of Storable objects to be stored.

Table Structure Similarly to the SQL implementation, there are several tables, some generated based on the SchemaSescription and other, control, tables exist regardless. Each table is represented in memory by a subclass of the impl.lmdb.common.tables.interfaces.LMDBTable trait, which provides utility methods such as transactional reads and computations.

Name	Key Type	Value Type	Description
Object Counter	Singleton	ObjId	Atomic counter holds the next ObjectId
Commit Counter	Singleton	CommitId	Atomic counter holds the next CommitId
View Counter	Singleton	ViewId	Atomic counter holds the next ViewId
Default table	Singleton	ViewId	Mutably stores the default view
Available Views	Singleton	Set[ViewId]	Mutably stores a set of available views
Views Table	ViewId	List[CommitId]	Maps a view to its constituent commits
Relations	(ObjectId, CommitId, RelationName)	Set[ObjId]	Single table stores all forward relations in a database
Reverse Relations	(ObjectId, CommitId, RelationName)	Set[ObjId]	Single table stores the reverse of all relations in the database

Singleton Keys denote tables with single key.

Name	Key Type	Value Type	Description
Retrieval Table	ObjectId	DBObject	Used to retrieve object data at end of a query
Empty Index	CommitId	ObjectId	Stores object Ids accessible a view. Used to compute Id˙A.
Column Index	(Commit, DBCell)	ObjectId	Provides an index over a field of a SchemaObject

For each SchemaObject in the Schemadescription, there exists a RetrievalTable, EmptyIndexTable, and a ColumnIndexTable per column in the SchemaObject.

Writes Writes occur in a similar way to the SQL implementation, again using commits to manage views. First we look up all the left and right hand side objects to find their ObjectIds (If needed, we create new entries in the retrieval table and update the index tables to insert the objects.), we then look up existing instances of the relations to insert and only insert new ones to the relation table with the new CommitId. Upon success, we create a new view and insert it to the views table and the available views table.

3.11.2 Original LMDB Implementation

impl.lmdb.original

The original LMDB backend implementation executed queries in much the same way as the in-memory backend, the only difference being the flat LMDB table structure. Interpretation occurs by a very similar to function that of the memory backend, instead passing around a list of Commits to search rather than the memory tree. At the end of a query's execution, the resulting ObjectIds are looked up in the relevant Retrieval table to extract the actual objects and convert them into user objects. Pathfinding and the repetition operators are handled in the same way as the memory backend by calling the relevant generic algorithms.

3.11.3 Batched

impl.lmdb.fast This is the first variant upon my simple original LMDB backend, it focuses on making small local optimisations rather than significant algorithmic changes.

Read Batching Firstly, the original implementation made suboptimal uses of reads. For example, when extracting objects at the end of query execution, it would look up each object individually with its own read transaction. This is sped up by batching together reads to separate keys and commits in the same transaction, allowing for a small speed up.

Pre-Caching The result of executing a FindSingle is constant throughout the whole query, so by lifting out and pre-emptively executing any FindSingles, we speed up execution by removing redundancy. More importantly, this helps enable the next operation.

FindFrom For the pathfinding commands, the transitive queries and From FindSingle query, we're not actually interested in finding all pairs matching a query, or interested in which pairs are related, but actually the reachability function $A \Rightarrow Set[B]$ (this is a kleisli arrow), which takes a node and finds the directly reachable neighbours. Since we do not have to do any bookwork to maintain the left hand side of any relation, this greatly simplifies our search. Hence, in these cases, we interpret the query using a new FindFrom method, which is an alternative interpreter. For example, joins now become a concatenation of these arrows (flatMap in Scala parlance), Id becomes the return function $(x \Rightarrow Set(x))$, Distinct becomes set subtraction as opposed to calling the Set.filter function, and, finally, we can make use of the SimpleFixedPointTraversal algorithms described above. The pre-caching above is useful here, since due to the flatMaps, the computation of FindSingles would be repeated many times otherwise.

3.11.4 Common Subexpression Elimination

impl.lmdb.cse A common performance issue in the previous implementation was that of redundancy and lack of general common subexpression elimination (CSE). When a common sub expression is repeated outside of an Upto or Exactly block, then it is repeatedly recalculated, leading to poor performance. In addition, there is lots of redundancy exposed by the above FindFrom optimisation. Consider calculation $Rjoin(SjoinT)$. If a node n appears in the results of $S(m)$, $S(m')$ for distinct $(?, m) \triangleleft_{A,B,v} R$, $(?, m') \triangleleft_{A,B,v} R$ (m, m' are right hand results of R), then $T(n)$ is computed redundantly in the overall calculation. These issues of redundancy can be solved using a combination of global CSE and memoization.

The Memoisation Problem When looking at a graph data structure, memoization is harder than when looking to optimise pure functions. To optimise a pure function $f : A \Rightarrow B$, we only need to store a hash table Map[A, B]. To compute $f(a)$, we simply look up a in the map. If it is in the map, we return the mapped value else, we call f, and add the result to the map before returning. In a large graph, computing and storing all pairs generated by a query is extremely wasteful, since many of those pairs might never be used (effectively, we would

have ignored the left-optimisation). Conversely, memoizing the full left-optimised function, $interpret(Q)\colon Set[A] \Rightarrow Set[(A, B)]$ is also wasteful, since there may be overlap in the sets used as keys. Instead, we memoise the FindFrom function of a query, $findFrom(Q)\colon A \Rightarrow Set[B]$, and reconstruct results from the pairs. This allows for the best overlap.

To make the best use of memoization, and to overusing memory by memoizing the same query twice, we need to also apply CSE to group together instances of the same query.

Retrievers A solution to these parameters is the use of an object called a Retriever. This exposes two methods which mirror those from interpreting a query.

```
trait RelationRetriever extends Logged {
     def find(from: Set[ObjId]): LMDBEither[Set[(ObjId, ObjId)]]
     def findFrom(from: ObjId): LMDBEither[Set[ObjId]]
}
```

For each subquery node in a query(e.g. And(Id˙A, Or(R˙1, R˙2))), we generate a retriever. Retrievers for primitive relations (e.g. Id˙A , Rel(R)) are uncached, as LMDB allows for very fast re-computations of these. In the other cases, we use a CachedRelationRetriever, which memorises an underlying lookup function. Using type-enrichment, I have implemented methods such as join, and, or, exactly, fixedPoint on RelationRetrievers, allowing them to be composed to mirror the query they are generated from. This memoises every node of the query to be executed, yielding a significant reduction of redundancy.

Monadic Compilation To avoid storing multiple RelationRetrievers for the same subexpression, it is useful to reuse the same retriever for every occurrence of a subquery. Hence when building a retriever for a query, we want to keep track of all subqueries we've seen before. This can be done using the monadic compilation pattern above. The compilation state stores a hashmap of all the subtrees that have already been computed (A bit like when constructing a binary decision diagram in automated theorem proving).

```
class QueryMemo(
    val pairs: Map[UnsafeFindPair, RelationRetriever],
    val singles: Map[UnsafeFindSingle, SingleRetriever]
)
```

Compilation state

The compilation function checks the memo for precomputed values, and if they are not already computed, recurses to compute subtrees, then composes the subtree results to get a new retriever for the current node, which is added to the compilation state.

3.11.5 Complex Common Subexpression Elimination

impl.lmdb.fastjoins Despite these optimisations, there are still sources of redundancy.

Index Building Firstly, the original CSE implementation uses the excessively generic join function to join result sets. This function wastes time by computing an index map on every call. So instead we can change the definition of a Retriever's public functions to mitigate the need to build indices.

```
trait RelationRetriever extends Logged {
  def find(from: Set[ObjId]): LMDBEither[Map[ObjId, Set[ObjId]]]
  def findFrom(from: ObjId): LMDBEither[Set[ObjId]]
}
```

Indeed, the function to join a pair of retrievers becomes a simpler flatmap of one map to another. Functions such as the union and intersection of retrievers also become simpler, (see impl.lmdb.fastjoins.retrievers.RelationRetriever.RelationRetrieverOps)

Exactly This implementation also explores other formulations of the Exactly(n, P) query. Exactly effectively joins together n repetitions of the underlying query in a linear fashion. This makes relatively poor reuse of the join function, since each join has different operand subqueries, so it cannot be memoised. This formulation does make good reuse of P, however.

Diagram showing n joins of P, linear

As joins over the set of results of query form a Monoid Link to appendix, we can use associativity to change the order in which the joins are evaluated, aiming for as much reuse as possible. This can be done in a manner similar to binary exponentiation (http://computingonline.net/computing/article/viewFile/229/204).

We calculate retrievers for P^{2^i} for each i less than the bitlength of n. We then join the relevant retrievers to to get a retriever for P^n. This only uses $O(Log_2(n))$ distinct joins, and makes very good reuse of the join functions. Diagram showing join structure;

```
def exactly(n: Int): RelationRetriever = repeat(n, outer, IdRetriever)

private def repeat(
    n: Int,
    acc: RelationRetriever,
    res: RelationRetriever
): RelationRetriever = if (n > 0) {
  val newRes = if((n & 1) == 1)  (res join acc) else res
  val newAcc = acc join acc
  repeat(n >> 1, newAcc, newRes)
} else res
```

Upto Optimisation We can also re-formulate an Upto(n, P) as an $Exactly(n, Or(P, Id_A))$ link to Proof in appendix, and hence use the same optimisation as above on Upto. The same cannot be said for FixedPoint, as we do not know statically at what point the underlying relation will converge, so cannot pick an n to compute Upto(n, P).

```
def upto(n: Int): RelationRetriever =
    repeat(n, outer or IdRetriever, IdRetriever)
```

Chapter 4

Evaluation

4.1 Unit Tests

The correctness of each backend is tested using a suite of 45 unit tests which verify adherence to semantics. These have wide range, testing over all the commands, all the possible ADT nodes, and the correct usage and separation of views. I have been using the regression test model, in that discovering a non-trivial bug, I've written a test case to target that bug, ensuring that it is not leaked into production again.

4.2 Performance Tests

In order to evaluate the effectiveness of the various optimisations to the LMDB backends described above, it was necessary to run performance tests over wide range of queries. The various LMDB backends were tested against each other as a control and particularly against the SQL backend as a standard to beat. The in-memory backend was omitted as initial tests indicated that it runs with approximately the same algorithmic characteristics as the Original LMDB implementation. The LMDB implementation was able to do indexing faster than Scala's tree based hash maps, which are used by the in-memory implementation. Further more, the original LMDB implementation is typically around 3x slower than the batched version for most jobs, so we also omit it for most tests, since most of the algorithms it uses are the same.

4.2.1 Hardware

Tests were run on the oslo machine belonging to Timothy Jones' group. Its specifications are shown below. All of the backends ran off of an SSD. Oslo Specs

4.2.2 Datasets

To evaluate tests on non-trivial examples, I sought to construct datasets over which large queries were feasible.

IMDB The first and most used collection of tests are derived from the IMDB movie database,the most popular 5000 of which were collected from kaggle https://www.kaggle.com/tmdb/tmdb-movie-metadata

The CSV data was processed using a python script (in src/resources/imdb/) into a simplified JSON format. A separate, larger dataset was constructed from tests for another graph database. https://github.com/arangodb/example-datasets/tree/master/Graphs/IMDB and manually (in python) converted to the same JSON format. I then wrote a Scala script which reads the JSON and writes the relations to a given database instance.

Slightly different parameters were given when generating each dataset.

Name	Movies Size	People Size	Notes
smallest	100	10351	100 movies with most actors
small	250	18167	250 movies with most actors
small ˙sparse	250	1029	250 movies with fewest actors
medium	1000	35626	1000 movies with most actors
medium˙sparse	1000	6310	1000 movies with fewest actors
large	12862	50134	Largest dataset

The objects in the database are as follows:

Objects

Name	Fields
Person	Name: String
Movie	Name: String Language: String
Date	Time: Long
Place	Place: String
Genre	Genre: String

Relations

Name	From	To
ActsIn	Person	Movie
Directed	Person	Movie
HasBirthday	Person	Date
BornIn	Person	Place
HasGenre	Movie	Genre

UFC A second dataset, built from UFC fight data, https://www.kaggle.com/cformey24/ufc-fights-data-1993-2232016/data was constructed in the same way to produce one JSON dataset.

Objects

Name	Fields
Person	Name: String, Height: Int, Weight: Int

Relations

Name	From	To	notes
Beat	Person	Person	
ShorterThan	Person	Person	Transitive reduction of the shorter than relation
LighterThan	Person	Person	Transitive reduction of the lighter than relation

The ShorterThan and LighterThan relations produce extremely sparse graphs which take a long time to converge under transitive closure. Size of graph.

4.2.3 Test Harness

In order to run tests against each other, I have written a typesafe test harness to run on oslo. This standardises the interface that individual test instances must implement. Each Test must have a setup method and a test method which is run on a DBInstance with the test method indexed by a TestIndex. The test specification must give a maximum index and a mask to avoid running inappropriate backends (For example, those that might take too long on a large test.) The benchmarks that I have run test only the read speed. The time taken to construct the database is not included.

4.2.4 Results

Overall Picture This needs to be more specific, rewrite An overall view of the results is that, as might be expected, the SQL implementation is the fastest, with the most aggressively optimised LMDB typically performing the closest in speed to SQL, calculate precise values though occasionally beating SQL's performance by up to 150%. Due to a high overhead of initiating individual queries, the SQL instance performed appalling on pathfinding queries. The LMDB optimisations worked very effectively where they were hypothesised to, such as queries with repeated joins and those which made use of Exactly and Upto, yielding orders of magnitude speedup. However, in some tests, the optimisations yielded overheads slowing down performance, especially those queries which make use of FindFrom rather than finding pairs.

Typically, the CSE optimisations alone do not provide a statistically significant speed up, or even slow down operations, however the optimisations also applied to joins and transitive queries do often show a large speedup.

The pathfinding implementation, for simplicity, sends queries at each stage of the breadth first search. This seems to particularly punish the SQL implementation, to the extent that it is around 2000x slower than the LMDB reference implementation. Stateful solutions to speed this up might include creating a temporary table in the database to store the subgraph being searched, but this complicates the SQL implementation significantly (clearing up on error becomes a lot harder). Hence, I am ignoring the pathfinding tests for the purpose of backend comparison.

Redundancy This test is to demonstrate the removal of redundancies in FindSingle commands. There is clearly a major increase in speed due to the introduction of CSE and memoisation. Since this is a FindSingle query, there is no usage of the optimised join formulations, so there is no speed gain by the most optimised LMDB backend. Postgres fails to fully optimise away the redundancy, even with use of the DISTINCT modifier of queries, and ends up running out of temporary space on this query.

Graph

Name	Redundancy	
Iterations	4 each	

Queries	`KevinBacon >> coactor`
	`KevinBacon >> (coactor -->--> (coactor -->--> coactor))`
	`KevinBacon >> (coactor -->--> (coactor -->--> (coactor -->--> coactor)))`
	Where `def coactor = ActsIn --><-- ActsIn`	

implementation	time (ms)	/optimised
RefImpl	381,972	1
LMDBCSE	379,280	0.993
LMDBatched	9,318,741	24.4
Postgres	Did not finish	Did not finish

Conjunctions and Disjunctions These two tests make use of the And and Or operators to combine several subqueries with some underlying common structure. Both resulted in similar performance patterns. The postgres implementation significantly outperformed all LMDB implementations, while the CSE related optimisations appeared to add overheads greater than the redundancy they removed. I expect the reason for this is that there is still not a large amount of redundancy exposed by the underlying sub queries, and that the implementation of unions and intersection in Scala sets is slower than the optimised C implementation used by postgres. In addition, constructing a conjunction and disjunction over the memoisation in the LMDB implementation also adds some overhead and reduces locality of reference. Graph

Name	Conjunctions	
Iterations	40 each	

Queries	`coactorWith(KevinBacon))`
	`coactorWith(KevinBacon) & coactorWith(TomCruise))`
	`coactorWith(KevinBacon) & coactorWith(TomCruise) & coactorWith(TomHanks))`
	Where
	`def coactorWith(a: Person) = ActsIn --> (a >> ActsIn) <-- ActsIn`	

implementation	time (ms)	/optimised
RefImpl	879,984	1
LMDBCSE	827,734	0.941
LMDBatched	791,678	0.900
Postgres	106,165	0.121

Graph

Name	Disjunctions	
Iterations	40 each	

Queries	`coactorWith(KevinBacon))`
	`coactorWith(KevinBacon) | coactorWith(TomCruise))`
	`coactorWith(KevinBacon) | coactorWith(TomCruise) | coactorWith(TomHanks))`
	Where
	`def coactorWith(a: Person) = ActsIn --> (a >> ActsIn) <-- ActsIn`	

implementation	time (ms)	/optimised
RefImpl	853,664	1
LMDBCSE	824,799	0.966
LMDBatched	800,455	0.938
Postgres	112,488	0.132

Tests that involve repetitions These are a suite of benchmarks for testing the optimisations and performance on repetitive queries. A general overview is that the join-reordering optimisations strongly sped up benchmarks which ran FindPairs queries, but in FindSingles queries, the overhead of this formulation did not compete as strongly with the fast SimpleFixedPointTraversal methods. SQL performed strongly in Upto queries which use built in, optimised, recursive CTEs to calculate results, but less strongly in Exactly queries which do not have a built-in formulation.

Exactly Test This test runs FindSingle Exactly queries of varying lengths over the small movies database. As explained, the SimpleFixedPointTraversal exactly method is faster than any gains by the join optimisations in the optimised LMDB version. Postgres also performed poorly due to the lack of a recursive CTE for self joins. Graph

Name	Exactly	
Iterations	50	

Queries	`TomHanks >> (`
	`((ActsIn -->(KevinBacon >> ActsIn))<-- ActsIn) * (index.i % 10)`
	`)`	

implementation	time (ms)	/optimised
RefImpl	1,752	1
LMDBCSE	1,798	1.03
LMDBatched	1,513	0.834
Postgres	102,461	58.5

Exactly Pairs This test instead runs a FindPair variable length Exactly query over the small movies database. Since this test involves keeping track of the root for each pair found, the Exactly-optimised LMDB implementation significantly outperforms the other LMDB implementations based on generic algorithm. Postgres also performs well in this test. Graph

Name	ExactlyPairs	
Iterations	50	
Queries	`((ActsIn -->(KevinBacon >> ActsIn))<-- ActsIn) * (index.i % 10))`	
implementation	time (ms)	/optimised
RefImpl	83,269	1
LMDBCSE	1,792,388	21.5
LMDBatched	1,787,390	21.5
Postgres	107,797	1.29

UptoTest This test runs Upto based FindPair queries over the large IMDB database. Here the upto-optimisation of the final LMDB backend provides significant speed-ups, such as 225x on the other LMDB backends and 1.5x over postgres. Graph

Name	Upto	
Iterations	10	
Queries	`((ActsIn -->(KevinBacon >> ActsIn))<-- ActsIn).*(0 --> index.i)`	
implementation	time (ms)	/optimised
RefImpl	126,448	1
LMDBCSE	28,491,910	225
LMDBatched	28,423,658	225
Postgres	197,880	1.56

UptoLarge Since the more naive LMDB instances took seven hours each to complete the above test, I re-ran the benchmark for only the reference and Postgres implementations for a longer period of time, yielding a similar ratio of performance. Graph

Name	Upto	
Iterations	10	
Queries	`((ActsIn -->(KevinBacon >> ActsIn))<-- ActsIn).*(0 --> (index.i % 10))`	
implementation	time (ms)	/optimised
RefImpl	1,059,091	1
Postgres	1,557,314	1.47

JoinSpeed This test is to demonstrate the cost of joins, we interleave queries that calculate two subqueries and then combine them using, firstly, a relatively fast And and, secondly, instead, a higher complexity join. In theory, the difference between these two types of queries should give us an idea of the cost of a join. Graph

Name	Join Speed	
Iterations	150 each	
Queries	`(ActsIn -><-- ActsIn) & (ActsIn -><-- ActsIn)` `(ActsIn -><-- ActsIn) -->--> (ActsIn -><-- ActsIn)`	
implementation	time (ms)	/optimised
RefImpl	368,320	1
LMDBCSE	525,992	1.43
LMDBatched	543,909	1.48
Postgres	289,352	0.786

More detail

Closing Thoughts These benchmarks provide some insights into the strengths and weaknesses of the various optimisations I have written. An option to produce a performant back-end might be to produce heuristics for estimating graph density and the relative performance of the various LMDB backends upon it to create an adaptive backend. Since all the LMDB backends use the same underlying database format (indeed the code for constructing and writing to the database remains the same), a backend could choose at query time which strategy it wants to use to execute each individual query.

Chapter 5

Conclusion

5.1 Successes

To the best of my knowledge, the graph database system I have built over the last seven months is one of the, if not the, first explicitly purely functional graph database systems in existence. This project is a demonstration that complex, performant software projects can be built using purely functional languages, whilst keeping the benefits of strong type systems. I feel that despite having written the project in Scala, a high level, often inefficient language I have achieved an acceptable level of performance for graph traversal. Although the system would need significant work in areas of security, networked access, and tests and tweaks on real world use cases to become commercially viable, it would be interesting to see if companies known for their use of functional programming, especially finance firms, might find use cases for graph databases

5.2 Further extensions

Interesting extensions to the core project that, given more time, I would have been interested to explore include

- An increase in the use of laziness to allow larger queries over larger graphs. Currently, the size of queries are limited by how large the result sets of sub-queries. However, the Postgres implementation typically runs out of temporary file space before this happens.
- The implementation of deletions and garbage collection of unused views of the database. Garbage collection in particular might be difficult, as the database is not required to store any state about client instances.
- Exploration of improvement of the schema system using a dependent types library, such as shapeless.
- Rewriting the lowest layer of the LMDB backends' interpreters in C or another low level language to extract maximum performance from the LMDB system, without having to construct Scala collections at every step.

5.3 Lessons learned

In hindsight, I might aimed to build a narrower project, focusing on a single backend and comparing performance against a widely used graph database such as Neo4j. Although I very much enjoyed the chance to build a large project in a purely functional style, implementing the large number of moving parts in the project distracted from implementing more interesting optimisations and really digging down to extract all redundancy.

5.4 Concluding Thoughts

I hope for another chance to explore this space as I feel there are still many areas worth investigating.

Bibliography

Appendix A

The Domain closure(A, v)

Do we split this? For the various semantics proofs it is necessary to define the Scott domain $closure(A, v)$ for some object type A and $View_\Sigma\ v$. This domain is the set of subsets x such that $[\![Id_A]\!](v) \subseteq x \subseteq A \times A$ with bottom element $\bot = [\![Id_A]\!](v)$ and partial order $x \sqsubseteq y \Leftrightarrow x \subseteq y$. From this point on, I shall use $x \subseteq y$ to mean $y \sqsubseteq x$.

$closure(A, v)$ **is a domain** Firstly, by definition,

$$\forall x.x \in closure(A, v) \Rightarrow x \supseteq [\![Id_A]\!](v)$$

hence $[\![Id_A]\!](v)$ is the bottom element.
Secondly for any chain $x_1 \subseteq x_2 \subseteq x_3 \subseteq ...$, $x_i \in closure(A, v)$, there exists a value $\bigsqcup_n x_n \in closure(A, v)$ such that $\forall i. \bigsqcup_n x_n \supseteq x_i$ and $\forall y.(\forall i.x_i \subseteq y) \Rightarrow y \supseteq \bigsqcup_n x_n$
proof: Take $\bigsqcup_n x_n = \bigcup_n x_n$. This is in $closure(A, v)$, since both $\bigcup_n x_n \supseteq [\![Id_A]\!](v)$ due to $\forall i.x_i \supseteq [\![Id_A]\!](v)$ by definition and $\bigcup_n x_n \subseteq A \times A$, by

$$\forall a.(a \in \bigcup_n x_n \wedge \neg(a \in A \times A)) \Rightarrow (\exists i.a \in x_i \wedge \neg a \in A \times A) \Rightarrow (\exists i.\neg x_i \subseteq A \times A)$$

yielding a contradiction if $\bigcup_n x_n \subseteq A \times A$ does not hold.
We know $\forall i. \bigcup_n x_n \supseteq x_i$ by definition, so it is an upper bound.
To prove it is a least upper bound, consider y such that $(\forall i.)y \supseteq x_i$

$$\begin{aligned} \forall a.(\exists i.a \in x_i) &\Rightarrow a \in y \\ \forall a. \bigvee_n (a \in x_n) &\Rightarrow a \in y \\ \forall a.(a \in \bigcup_n x_n) &\Rightarrow a \in y \\ \therefore \bigcup_n x_n &\subseteq y \\ \Box & \end{aligned} \tag{A.1}$$

Appendix B

Correspondence of operational and denotational semantics

neaten In order to use the denotational semantics to construct an interpreter or compiler we need to prove they are equivalent to the operational semantics. Namely:

For any pair query P, schema Σ and $View_\Sigma$ v:

$$\Sigma \vdash P\colon (A,B) \Rightarrow (a,b \lhd_{(A,B),v} P \Leftrightarrow (a,b) \in [\![P]\!](v))$$

And for any single query S, schema Σ and $View_\Sigma$ v:

$$\Sigma \vdash S\colon A \Rightarrow (a \lhd_{A,v} S \Leftrightarrow a \in [\![S]\!](v))$$

In order to prove these two propositions, we define two induction hypotheses

$$\Phi(\Sigma,P,A,B) \Leftrightarrow (\Sigma \vdash P\colon (A,B) \Rightarrow \forall v \in View_\Sigma, (a,b) \in A{\times}B.((a,b) \lhd_{(A,B),v} P) \Leftrightarrow (a,b) \in [\![P]\!](v))))$$

$$\Psi(\Sigma,S,A) \Leftrightarrow (\Sigma \vdash S\colon A \Rightarrow \forall v \in View_\Sigma, a \in A.(a \lhd_{A,v} S) \Leftrightarrow (a \in [\![S]\!](v)))$$

Now we shall induct over the structures of P and S starting with the SingleQuery cases

Case $S = AndS(S',S'')$

$$\begin{aligned} a \in [\![S]\!](v) &\Leftrightarrow a \in ([\![S']\!](v) \cap [\![S'']\!](v)) \\ &\Leftrightarrow (a \in [\![S']\!](v) \wedge a \in [\![S'']\!](v)) \\ &\Leftrightarrow (a \lhd_{A,v} S' \wedge a \lhd_{A,v} S'') \text{ by } \Psi(\Sigma,S',A),\ \Psi(\Sigma,S'',A) \\ &\Leftrightarrow a \lhd_{A,v} AndS(S',S'') \text{ by inversion of (AndS)} \end{aligned} \tag{B.1}$$

Case $S = OrS(S',S'')$

$$\begin{aligned} a \in [\![S]\!](v) &\Leftrightarrow a \in ([\![S']\!](v) \cup [\![S'']\!](v)) \\ &\Leftrightarrow (a \in [\![S']\!](v) \vee a \in [\![S'']\!](v)) \\ &\Leftrightarrow (a \lhd_{A,v} S' \vee a \lhd_{A,v} S'') \text{ by } \Psi(\Sigma,S',A),\ \Psi(\Sigma,S'',A) \\ &\Leftrightarrow a \lhd_{A,v} OrS(S',S'') \text{ by inversion of (OrS)} \end{aligned} \tag{B.2}$$

Case $S = From(S',P)$

$$\begin{aligned} b \in [\![S]\!](v) &\Leftrightarrow \exists a \in A. \quad (a \in [\![S']\!](v) \wedge (a,b) \in [\![[\![P]\!](v)]\!](v)) \\ &\Leftrightarrow \exists a \in A. \quad (a \lhd_{A,v} S' \wedge (a,b) \lhd_{(A,B),v} P) \text{ by } \Psi(\Sigma,S',A),\ \Phi(\Sigma,P,A,B) \\ &\Leftrightarrow b \lhd_{B,v} From(S',P) \quad \text{by inversion of (OrS)} \end{aligned} \tag{B.3}$$

Case $S = Find(f)$

$$\begin{aligned} a \in [\![S]\!](v) &\Leftrightarrow a \in v(A) \wedge f(a) \downarrow True \qquad \text{by inversion of the type rule (Find)} \\ &\Leftrightarrow a \lhd_{A,v} Find(f) \text{ by definition} \end{aligned} \tag{B.4}$$

Now, looking at the FindPair queries

Case $P = Rel(r)$

$$\begin{aligned} (a,b) \in [\![P]\!](v) &\Leftrightarrow (a,b) \in v(r) \\ &\Leftrightarrow (a,b) \lhd_{(A,B),v} Rel(r) \text{ by definition} \end{aligned} \tag{B.5}$$

Case $P = RevRel(r)$

$$\begin{aligned} (a,b) \in [\![P]\!](v) &\Leftrightarrow (b,a) \in v(r) \\ &\Leftrightarrow (a,b) \lhd_{(A,B),v} RevRel(r) \text{ by definition} \end{aligned} \tag{B.6}$$

Case $P = Id_A$

$$\begin{aligned} (a,b) \in [\![P]\!](v) &\Leftrightarrow a \in v(A) \wedge a = b \\ &\Leftrightarrow (a,b) \lhd_{(A,B),v} Id_A \text{ by definition} \end{aligned} \tag{B.7}$$

Case $P = Chain(P', Q)$

We have by inversion of the (Chain) type rule $\Sigma \vdash P\colon (A,C) \Leftrightarrow \exists B. \quad \Sigma \vdash P'\colon (A,B) \wedge \Sigma \vdash Q\colon (B,C)$

$$\begin{aligned} (a,c) \in [\![P]\!](v) &\Leftrightarrow (a,c) \in join([\![P']\!](v), [\![Q]\!](v)) \\ &\Leftrightarrow \exists b \in B. \quad (a,b) \in [\![P']\!](v) \wedge (b,c) \in [\![Q]\!](v) \\ &\Leftrightarrow \exists b \in B. \quad (a,b) \lhd_{(A,B),v} P' \wedge (b,c) \lhd_{(B,C),v} Q \qquad \text{by } \Phi(\Sigma, P', A, B),\ \Phi(\Sigma, Q, B, C) \\ &\Leftrightarrow (a,c) \lhd_{(A,C),v} Chain(P', Q) \text{ by definition} \end{aligned} \tag{B.8}$$

Case $P = And(P', Q)$

$$\begin{aligned} (a,b) \in [\![P]\!](v) &\Leftrightarrow (a,b) \in ([\![P']\!](v) \cap [\![Q]\!](v)) \\ &\Leftrightarrow (a,b) \in [\![P']\!](v) \wedge (a,b) \in [\![Q]\!](v) \\ &\Leftrightarrow (a,b) \lhd_{(A,B),v} P' \wedge (a,b) \lhd_{(A,B),v} Q \qquad \text{by } \Phi(\Sigma, P', A, B),\ \Phi(\Sigma, Q, A, B) \\ &\Leftrightarrow (a,b) \lhd_{(A,B),v} And(P', Q) \text{ by inversion of (And)} \end{aligned} \tag{B.9}$$

Case $P = Or(P', Q)$

$$\begin{aligned} (a,b) \in [\![P]\!](v) &\Leftrightarrow (a,b) \in ([\![P']\!](v) \cup [\![Q]\!](v)) \\ &\Leftrightarrow (a,b) \in [\![P']\!](v) \vee (a,b) \in [\![Q]\!](v) \\ &\Leftrightarrow (a,b) \lhd_{(A,B),v} P' \vee (a,b) \lhd_{(A,B),v} Q \qquad \text{by } \Phi(\Sigma, P', A, B),\ \Phi(\Sigma, Q, A, B) \\ &\Leftrightarrow (a,b) \lhd_{(A,B),v} Or(P', Q) \text{ by inversion of (Or1), (Or2)} \end{aligned} \tag{B.10}$$

Case $P = AndLeft(P', S)$

$$\begin{aligned} (a,b) \in [\![P]\!](v) &\Leftrightarrow (a,b) \in [\![P']\!](v) \wedge a \in [\![S]\!](v) \\ &\Leftrightarrow (a,b) \lhd_{(A,B),v} P' \wedge a \lhd_{A,v} S \qquad \text{by } \Phi(\Sigma, P', A, B),\ \Psi(\Sigma, S, A) \\ &\Leftrightarrow (a,b) \lhd_{(A,B),v} AndLeft(P', S) \text{ by inversion of (AndLeft)} \end{aligned} \tag{B.11}$$

Case $P = AndRight(P', S)$

$$\begin{aligned}(a,b) \in [\![P]\!](v) &\Leftrightarrow (a,b) \in [\![P']\!](v) \wedge b \in [\![S]\!](v) \\ &\Leftrightarrow (a,b) \lhd_{(A,B),v} P' \wedge b \lhd_{B,v} S \quad \text{by } \Phi(\Sigma, P', A, B),\ \Psi(\Sigma, S, B) \\ &\Leftrightarrow (a,b) \lhd_{(A,B),v} AndRight(P', S) \text{ by inversion of (AndRight)}\end{aligned} \tag{B.12}$$

Case $P = Distinct(P')$

$$\begin{aligned}(a,b) \in [\![P]\!](v) &\Leftrightarrow (a,b) \in [\![P']\!](v) \wedge a \neq b \\ &\Leftrightarrow (a,b) \lhd_{(A,B),v} P' \wedge a \neq b \quad \text{by } \Phi(\Sigma, P', A, B) \\ &\Leftrightarrow (a,b) \lhd_{(A,B),v} Distinct(P') \text{ by inversion of (AndRight)}\end{aligned} \tag{B.13}$$

Case $P = Exactly(n, P')$
We have by inversion of the (Exactly) type rule $\Sigma \vdash P\colon (A, A) \wedge \Sigma \vdash P'\colon (A, A)$

$$\text{let } f = (\lambda pairs.join([\![P']\!](v), pairs)) \tag{B.14}$$
$$\text{then}$$

$$(a,b) \in [\![P]\!](v) \Leftrightarrow (a,b) \in f^n[\![Id_A]\!](v) \tag{B.15}$$
$$\text{hence it suffices to prove}$$

$$(a,b) \in f^n[\![Id_A]\!](v) \Leftrightarrow (a,b) \lhd_{(A,A),v} Exactly(n, P') \tag{B.16}$$

Case $Exactly(0, P')$:

$$f^0[\![Id_A]\!](v) = [\![Id_A]\!](v)$$
$$\text{so by } \Phi(\Sigma, (a,b), A, A), Id_A)$$

$$\begin{aligned}(a,b) \in f^0[\![Id_A]\!](v) &\Leftrightarrow (a,b) \in [\![Id_A]\!](v) \\ &\Leftrightarrow (a,b) \lhd_{(A,A),v} Id_A \\ &\Leftrightarrow (a,b) \lhd_{(A,A),v} Exactly(0, P')\end{aligned} \tag{B.17}$$

Case $Exactly(n+1, P')$**, assuming** $\Phi(\Sigma, Exactly(n, P'), A, A)$**:**

$$f^{n+1}[\![Id_A]\!](v) = f(f^n([\![Id_A]\!](v)))$$
$$\text{so by } \Phi(\Sigma, Exactly(n, P'), A, A)$$

$$\begin{aligned}(a,b) \in f^{n+1}[\![Id_A]\!](v) &\Leftrightarrow \exists a'.(a,a') \in [\![P']\!](v) \wedge (a',b) \in f^n([\![Id_A]\!](v)) \quad \text{by definition of } join \text{ and } f \\ &\Leftrightarrow (a,a') \lhd_{(A,A),v} P \wedge (a',b) \lhd_{(A,A),v} Exactly(n,p) \\ &\Leftrightarrow (a,b) \lhd_{(A,A),v} Exactly(n+1, P') \text{ by (Exactly n+1)}\end{aligned} \tag{B.18}$$

Case $P = Upto(n, P')$
We have by inversion of the (Upto) type rule $\Sigma \vdash P\colon (A, A) \wedge \Sigma \vdash P'\colon (A, A)$

$$\text{let } f = (\lambda pairs.join([\![P']\!](v), pairs) \cup pairs) \tag{B.19}$$
$$\text{then}$$

$$[\![P]\!](v) = f^n[\![Id_A]\!](v) \text{We now case split on n} \tag{B.20}$$

Case $Upto(0, P')$

$$\begin{aligned}(a,b) \in [\![Upto(0,P')]\!](v) &\Leftrightarrow (a,b) \in f^0[\![Id_A]\!](v)\\ &\Leftrightarrow (a,b) \in [\![Id_A]\!](v)\\ &\Leftrightarrow (a,b) \lhd_{(A,A),v} Id_A \text{ by } \Phi(\Sigma, Id_A, A, A)\\ &\Leftrightarrow (a,b) \lhd_{(A,A),v} Upto(0,P') \text{ by (Upto0)}\end{aligned} \tag{B.21}$$

Case $Upto(n+1, P')$

$$\begin{aligned}(a,b) \in [\![Upto(m+1,P')]\!](v) &\Leftrightarrow (a,b) \in f^{m+1}[\![Id_A]\!](v)\\ &\Leftrightarrow (a,b) \in (join([\![P']\!](v), f^m[\![Id_A]\!](v)) \cup f^m[\![Id_A]\!](v))\\ &\Leftrightarrow (a,b) \in join([\![P']\!](v), [\![Upto(m,P)]\!](v)) \vee (a,b) \in [\![Upto(m,P')]\!](v)\\ &\Leftrightarrow (\exists a'.(a,a') \in [\![P']\!](v) \wedge (a',b) \in [\![Upto(m,P')]\!](v)) \vee (a,b) \lhd_{(A,A),v} Upto(m,P')\\ &\Leftrightarrow (\exists a'.(a,a') \lhd_{(A,A),v} P' \wedge (a',b) \lhd_{(A,A),v} Upto(m,P')) \vee (a,b) \lhd_{(A,A),v} Upto(m,P')\\ &\Leftrightarrow (a,b) \lhd_{(A,A),v} Upto(m+1,P') \text{ by (Upto n+1), (Upto n)}\end{aligned} \tag{B.22}$$

case $P = FixedPoint(P')$

We have by inversion of the (FixedPoint) type rule $\Sigma \vdash P\colon (A,A) \wedge \Sigma \vdash P'\colon (A,A)$

$$\begin{aligned}&\text{let } f = (\lambda pairs.join([\![P']\!](v), pairs) \cup pairs)\\ &\qquad\qquad \text{then}\\ &[\![P]\!](v) = fix(f) \text{ In the domain } closure(A,v)\end{aligned} \tag{B.23}$$

Lemma: f is continuous in the domain $closure(A, v)$

Firstly, f is monotonous.
Let $x \subseteq y$

$$\begin{aligned}(a,b) \in f(x) &\Rightarrow (a,b) \in x \vee (\exists a'.(a,a') \in [\![P']\!](v) \wedge (a',b) \in x)\\ &\Rightarrow (a,b) \in y \vee (\exists a'.(a,a') \in [\![P']\!](v) \wedge (a',b) \in y)\\ &\Rightarrow (a,b) \in f(y)\\ \therefore f(x) &\subseteq f(y)\end{aligned} \tag{B.24}$$

Secondly, f **preserves the** lub**s of chains.**
Consider a chain $x1 \subseteq x2 \subseteq ...$ in $closure(A, v)$ Since $closure(A, v)$ is a domain, the lub, $\bigcup_n x_n$ is also in $closure(A, v)$

$$\begin{aligned}\forall m.x_m &\subseteq \bigcup_n x_n\\ \forall m.f(x_n) &\subseteq f(\bigcup_n (x_n))\\ \therefore \bigcup_n f(x_n) &\subseteq f(\bigcup_n (x_n))\end{aligned} \tag{B.25}$$

To get the inverse relation,

$$(a,b) \in f(\bigcup_n (x_n)) \Rightarrow ((\exists n.(a,b) \in x_n) \vee (\exists m, a'.(a,a') \in [\![P']\!](v) \wedge (a',b) \in x_m) \tag{B.26}$$

let $n' = max(n, m)$ so $x_n \subseteq x_{n'} \wedge x_m \subseteq x_{n'}$

$$\begin{aligned} &\exists n'.((a,b) \in x_{n'}) \vee (\exists a'.(a,a') \in [\![P']\!](v) \wedge (a',b) \in x_{n'}) \\ &\therefore \exists n'.(a,b) \in f(x_{n'}) \\ &\therefore \exists n'.f(\bigcup_n x_n) \subseteq f(x_{n'} \\ &\therefore f(\bigcup_n x_n) \subseteq \bigcup_n f(x_{n'}) \end{aligned} \tag{B.27}$$

So f is Scott-continuous.
Now, by Tarski's fixed point theorem

$$[\![FixedPoint(P')]\!](v) = fix(f) = \bigsqcup_n f^n(\bot)$$

Lemma $(a,b) \lhd_{(A,A),v} FixedPoint(P') \Leftrightarrow \exists n.(a,b) \in f^n(\bot)$ Firstly, in the forwards direction, $(a,b) \lhd_{(A,A),v} FixedPoint(P') \Rightarrow \exists n.(a,b) \in f^n(\bot)$

By inversion of the operational rules (FixedPoint0), (FixedPoint n) and $(a,b) \lhd_{(A,A),v} FixedPoint(P')$, we get two cases.

Case $(a,b) \lhd_{A,A,v} Id_A$**:**
by $\Phi(\Sigma, Id_A, A, A), (a,b) \in [\![Id_A]\!](v) = \bot$
so $n = 0$

Case $(a,b) \lhd_{(A,A),v} P' \wedge (b,c) \lhd_{(A,A),v} FixedPoint(P')$
(hence $(a,c) \lhd_{(A,A),v} FixedPoint(P')$)
by $\Phi(\Sigma, P', A, A), and (b,c) \lhd_{(A,A),v} FixedPoint(P')$

$$(a,b) \in [\![P']\!](v) \wedge \exists n.(b,c) \in f^n(\bot)$$

Instantiating with $m = n$ gives

$$(a,b) \in [\![P']\!](v) \wedge (b,c) \in f^m(\bot)$$

so $(a,c) \in f(f^m(\bot)) = f^{m+1}(\bot)$
Hence $(a,b) \lhd_{(A,A),v} FixedPoint(P') \Rightarrow \exists n.(a,b) \in f^n(\bot)$

To go the other way, we need to prove $(a,b) \in \bigsqcup_n f^n(\bot) \Rightarrow (a,b) \lhd_{(A,A),v} FixedPoint(P')$
$(a,b) \in \bigsqcup_n f^n(\bot)$ Means that either:

Case $(a,b) \in \bot$

$$\begin{aligned} &\therefore (a,b) \in [\![Id_A]\!](v) \\ &\therefore (a,b) \lhd_{(A,A),v} Id_A \text{ By } \Phi(\Sigma, Id_a, A, A) \\ &\therefore (a,b) \in [\![Id_A]\!](v) \text{ By (Fix1)} \end{aligned} \tag{B.28}$$

Case $\exists n \geq 0.(a,b) \in f^{n+1}(\bot) \wedge \neg((a,b) \in f^n(\bot))$

$$\begin{aligned} &(\exists a'.(a,a') \in [\![P']\!](v) \wedge (a',b) \in f^n(\bot)) \vee (a,b) \in f^n(\bot) \wedge \neg((a,b) \in f^n(\bot)) \\ &\qquad \therefore \exists a'.(a,a') \in [\![P']\!](v) \wedge (a',b) \in f^n(\bot) \\ &\qquad \therefore (a,a') \lhd_{(A,A),v} P' \wedge (a',b) \lhd_{(A,A),v} FixedPoint(P') \\ &\qquad \therefore (a,b) \lhd_{(A,A),v} FixedPoint(P') \end{aligned} \tag{B.29}$$

so we have $(a,b) \lhd_{(A,A),v} FixedPoint(P') \Leftrightarrow \exists n.(a,b) \in f^n(\bot) \Leftrightarrow (a,b) \in [\![FixedPoin(P')]\!](v)$

□

Appendix C

Joins on $Query(v)$ as a monoid

In order to justify some of our optimisations, we need to know certain properties of join denotation.

Firstly, we want to define establish closure and associatiavity of the *join* function defined in the denotational semantics.

We want to define these for queryable subsets of a view $v \in View_\Sigma$

$$Query(v) = \{s \mid \exists P, A, B.s = [\![P]\!](v) \wedge \Sigma \vdash P\colon A, B\} \tag{C.1}$$

C.1 Lemma: Join is closed on $Query(v)$

Proof: For $s, t \in Query(v)$ there exists P, Q, A, B, C, D such that $\Sigma \vdash P\colon A, B$ and $\Sigma \vdash Q\colon C, D$ hold and $s = [\![P]\!](v) \wedge t = [\![Q]\!](v)$

Case $B \neq C$ Then the join is empty, since no $b \in B = c \in C$, so

$$join([\![P]\!](v), [\![Q]\!](v)) = \emptyset = [\![Distinct(Id_T)]\!](v) \tag{C.2}$$

For some type $T \in \Sigma$ (We know Σ at least one type, since the typings of P and Q hold.)

Case $B = C$ Then the join is not empty. By the correspondence of denotational and operational semantics, we have

$$join([\![P]\!](v), [\![Q]\!](v)) = [\![Chain(P, Q)]\!](v) \tag{C.3}$$

with the following typing

$$\Sigma \vdash Chain(P, Q)\colon A, D \tag{C.4}$$

C.2 Lemma: Join is associative on $Query(v)$

Proof: For $r, s, t \in Query(v)$ there exist queries P, Q, R and object types A, B, C, D, E, F such that $\Sigma \vdash P\colon A, B$, $\Sigma \vdash Q\colon C, D$, and $\Sigma \vdash R\colon E, F$ hold and $r = [\![P]\!](v) \wedge s = [\![Q]\!](v) \wedge t = [\![R]\!](v)$ We want to prove that $join(r, join(s, t)) = join(join(r, s), t)$

Case $B \neq C$ Then

$$\begin{aligned} join(r, join(s, t)) &= join(r, u) \text{ For some } u, \text{ either } u = \emptyset \text{ or } u \text{ has a left type of } C \\ &= \emptyset \text{ hence equals the empty set} \\ &= join(\emptyset, t) \\ &= join(join(r, s), t) \end{aligned} \tag{C.5}$$

Case $D \neq E$ Then similarly

$$\begin{aligned} join(join(r,s),t) &= join(u,t) \text{ For some } u, \text{ either } u = \emptyset \text{ or } u \text{ has a right type of } D \\ &= \emptyset \text{ hence equals the empty set} \\ &= join(r,\emptyset) \\ &= join(r, join(s,t)) \end{aligned} \tag{C.6}$$

Case $B = C$ **and** $D = E$ Then

$$\begin{aligned} (a,f) \in join(r, join(s,t)) &\Leftrightarrow (a,f) \in join([\![P]\!](v), join([\![Q]\!](v), [\![R]\!](v))) \\ &\Leftrightarrow a, f \lhd_{A,F,v} Chain(P, Chain(Q,R)) \text{ by deno-oper correspondence} \\ &\Leftrightarrow \exists b,d.(a,b) \lhd_{A,B,v} P \wedge (b,d) \lhd_{B,D,v} Q \wedge (d,f) \lhd_{D,F,v} Q \\ &\Leftrightarrow (a,f) \lhd_{A,F,v} Chain(Chain(P,Q),R) \\ &\Leftrightarrow (a,f) \in [\![Chain(Chain(P,Q),R)]\!](v) \\ &\Leftrightarrow (a,f) \in join(join([\![P]\!](v), [\![Q]\!](v)), [\![R]\!](v)) \\ &\Leftrightarrow (a,f) \in join(join(r,s),t) \end{aligned} \tag{C.7}$$

Hence $Query(v)$ is a monoid with *join*. However, it doesn't particularly stick to a nice typescheme.

C.3 Joins on $Query_A(v)$ as a monoid

If we now take the more useful subset $Query_A(v)$ of $Query(v)$

$$Query_A(v) = \{s \subseteq A \times A \mid \exists P.\Sigma \vdash P\colon A, A \wedge s = [\![P]\!](v)\} \tag{C.8}$$

Trivially, from the above, *join* over $Query_A(v)$ also forms a monoid.
This enables several optimisations.
Firstly, writing p for $[\![P]\!](v)$ and with the binary representation of $n = \sum_i b_i * 2^i$

$$\begin{aligned} [\![Exactly(n,P)]\!](v) &= p^n \text{ In } Query_A(v) \\ &= \prod_i p^{b_i * 2^i} \end{aligned} \tag{C.9}$$

Todo: join distributes over and and or

C.4 Joins distribute over Or

It is useful to know how *join* interacts with other queries. Specifically we can do some re-writing if it is the case that join distributes over *Or*

$$[\![Chain(P, Or(Q,R))]\!](v) = [\![Or(Chain(P,Q), Chain(P,R))]\!](v) \tag{C.10}$$

Firstly, we have by inversion of the type rules for *Chain* and *Or* that:

$$\begin{aligned} \Sigma \vdash Chain(P, Or(Q,R))\colon A, C &\Leftrightarrow \Sigma \vdash P\colon A, B \wedge \Sigma \vdash Q, R\colon B, C \\ &\Leftrightarrow \Sigma \vdash Chain(P,Q)\colon A, C \wedge \Sigma \vdash Chain(P,R)\colon A, C \\ &\Leftrightarrow \Sigma \vdash Or(Chain(P,Q), Chain(P,R))\colon A, C \end{aligned} \tag{C.11}$$

$$\begin{aligned} [\![Chain(P, Or(Q,R))]\!](v) &= \{(a,c) \mid \exists b.(a,b) \in [\![P]\!](v) \wedge (b,c) \in ([\![Q]\!](v) \cup [\![R]\!](v)\}) \\ &= \{(a,c) \mid \exists b.((a,b) \in [\![P]\!](v) \wedge (b,c) \in [\![Q]\!](v)) \vee ((a,b) \in [\![P]\!](v) \wedge (b,c) \in [\![R]\!](v))\} \\ &= [\![Chain(P,Q)]\!](v) \cup [\![Chain(P,R)]\!](v) \\ &= [\![Or(Chain(P,Q), Chain(P,R))]\!](v) \end{aligned} \tag{C.12}$$

C.5 $Upto(n,P)$ expressed as $Exactly(n,P')$

Thanks to the previous section, we can now rewrite the denotation of *upto*

$$\begin{aligned}[\![Upto(n,P)]\!](v) &= (\lambda pairs.join([\![P]\!](v), pairs) \cup pairs)^n [\![Id_A]\!](v) \\ &= (\lambda pairs.join([\![P]\!](v) \cup [\![Id_A]\!](v), pairs)^n [\![Id_A]\!](v) \text{ Since } Id_A \text{ is the identity of join} \\ &= [\![Exactly(n, Or(P, Id_A))]\!](v)\end{aligned} \tag{C.13}$$

This means that we can evaluate *Upto* queries as *Exactly* queries, and apply the binary-representation driven construction above to evaluate queries using fewer unique joins.

C.6 Joins do not distribute over And

Consider the schema containing object types A, B, C and relations $R_1{:}\,A,B$, $R_2{:}\,B,C$, and $R_3{:}\,B,C$ and a view v in this schema, with relations $R_1 \mapsto \{(a,b1),(a,b2)\}$, $R_2 \mapsto \{(b1,c)\}$, and $R_3 \mapsto \{(b2,c)\}$.

Clearly

$$(a,c) \in [\![And(Chain(R_1,R_2), Chain(R_1,R_3))]\!](v) \tag{C.14}$$

But

$$[\![And(R_2,R_3)]\!](v) = \emptyset \tag{C.15}$$

So

$$[\![Chain(R_1, And(R_2,R_3))]\!](v) = \emptyset \tag{C.16}$$

Appendix D

Scala Algebraic Data Type Definitions

Here, I have attached the definitions of the Scala typed intermediate representation. Note how the use of generic type parameters causes the Scala level type checking to also do the query-language type checking and inference. Also note how the use of SchemaObject[A] implicit parameters forces membership of the type-class of types that can be stored in the database.

```
sealed abstract class FindPair[A, B](
    implicit val sa: SchemaObject[A],
    val sb: SchemaObject[B]
) {
  /**
    * Reverse the relation
    */
  def reverse: FindPair[B, A]

  /**
    * Erase the type of the object, wrt SchemaDescription
    */
  def getUnsafe(sd: SchemaDescription):  MissingRelation \/ UnsafeFindPair
}
```

```
sealed abstract class FindSingle[A](implicit val sa: SchemaObject[A]) {
  /**
    * Convert to unsafe
    */
  def getUnsafe(sd: SchemaDescription): MissingRelation \/ UnsafeFindSingle
}
```

Definitions of FindPair case classes. For the sake of brevity and clarity, I have removed the implementations of reverse and getUnsafe. In short, they were defined straightforwardly to recursively process the tree.

```
/**
  * Search for pairs related by a [[Relation]]
  */
case class Rel[A, B](r: Relation[A, B])
```

```
    (implicit sa: SchemaObject[A], sb: SchemaObject[B])
    extends FindPair[A, B]

/**
  * Search for pairs related in reverse by a [[Relation]]
  */
case class RevRel[A, B](r: Relation[B, A])
    (implicit sa: SchemaObject[A], sb: SchemaObject[B])
    extends FindPair[A, B]

/**
  * Search for pairs that appear in the result of both subexpressions
  */

case class And[A, B](left: FindPair[A, B], right: FindPair[A, B])
    (implicit sa: SchemaObject[A], sb: SchemaObject[B])
    extends FindPair[A, B]

/**
  * Search pairs in the result of the left subexpression,
  * such that the right of the pair is in result of the
  * right subexpression
  */
case class AndRight[A, B](left: FindPair[A, B], right: FindSingle[B])
    (implicit sa: SchemaObject[A], sb: SchemaObject[B])
    extends FindPair[A, B]

/**
  * Search for pairs of the left sub expression
  * such that the left of the pair is in the result
  * of the right subexpression
  */
case class AndLeft[A, B](left: FindPair[A, B], right: FindSingle[A]
    (implicit sa: SchemaObject[A], sb: SchemaObject[B])
    extends FindPair[A, B]


/**
  * Search for pairs that appear in the result of either subexpression
  */
case class Or[A, B](left: FindPair[A, B], right: FindPair[A, B])
    (implicit sa: SchemaObject[A], sb: SchemaObject[B])
    extends FindPair[A, B]

/**
  * search for pairs (a, c) such that there
  * exists b, (a, b) is in the result of left, (b, c)
  * is in the result of right
  */

case class Chain[A, B, C](
                           left: FindPair[A, B],
                           right: FindPair[B, C]
                         )
    (implicit sa: SchemaObject[A], sb: SchemaObject[B], sc: SchemaObject[C])
    extends FindPair[A, C]

/**
  * return a set of repeated pairs of the type
  *
  * Chain(a, id) = Chain(id, a) = a
  */
```

```
case class FindIdentity[A]()(implicit sa: SchemaObject[A])
extends FindPair[A, A]

/**
  * filter the result of the sub expression for those
  * pairs which do not equal each other
  */

case class Distinct[A, B](rel: FindPair[A, B])
    (implicit sa: SchemaObject[A], sb: SchemaObject[B])
extends FindPair[A, B]

/**
  * Find pairs that are related by n repetitions
  * of the underlying subexpression
  */

case class Exactly[A](n: Int, rel: FindPair[A, A])
    (implicit sa: SchemaObject[A])
    extends FindPair[A, A]

/**
  * Find pairs that are related by up to n
  * repetitions of the underlying subexpression
  */

case class Upto[A]( n: Int, rel: FindPair[A, A])
    (implicit sa: SchemaObject[A])
    extends FindPair[A, A]

/**
  * Find pairs that are related by the transitive closure
  * of the underlying subexpression
  */
case class FixedPoint[A](rel: FindPair[A, A])
    (implicit sa: SchemaObject[A])
    extends FindPair[A, A]
```

Definition of FindSingle case classes. Again, I have ommitted the implementations of getUnsafe for the sake of brevity

```
/**
  * Lookup a findable
  */
case class Find[A](pattern: Findable[A])
    (implicit sa: SchemaObject[A])
    extends FindSingle[A]
/**
  * Narrow down a findSingle query
  */
case class AndS[A
    (left: FindSingle[A], right: FindSingle[A])(implicit sa: SchemaObject[A])
    extends FindSingle[A]
/**
  * Find objects b: B, such that there exists a: A
  * in the result of start that (a, b) is in the result of rel
  */
```

```
case class From[A, B](start: FindSingle[A], rel: FindPair[A, B])
    (implicit sa: SchemaObject[A], sb: SchemaObject[B])
    extends FindSingle[B]

/**
  * Broaden a findSingle query
  */
case class OrS[A](left: FindSingle[A], right: FindSingle[A])
    (implicit sa: SchemaObject[A])
    extends FindSingle[A]
```

Appendix E

Denotational Semantics Based Memory Implementation

E.1 FindPairs

The simplistic denotational semantics based Memory interpreter

```
def findPairsSetImpl(
    t: UnsafeFindPair,
    left: Set[MemoryObject],
    tree: MemoryTree
): MemoryEither[Set[RelatedPair]] = {


    def recurse(
        t: UnsafeFindPair,
        left: Set[MemoryObject]
    ) = findPairsSetImpl(t, left, tree)


    t match {
      case USAnd(l, r) =>
        for {
        leftRes <- recurse(l, left)
        rightRes <- recurse(r, left)
      } yield leftRes.intersect(rightRes)


      case USAndRight(l, r) => for {
        leftRes <- recurse(l, left)
        rightRes <- findSingleSetImpl(r, tree)
      } yield leftRes.filter{case (a, b) => rightRes.contains(b)}

      case USAndLeft(p, s) => for {
        rightRes <- findSingleSetImpl(s, tree)
        pairRes <- recurse(p, rightRes intersect left)
      } yield pairRes


      case USOr(l, r) => for {
        leftRes <- recurse(l, left)
        rightRes <- recurse(r, left)
      } yield leftRes.union(rightRes)

      case USChain(l, r) => for {
        lres <- recurse(l, left)
```

```
        rres <- recurse(r, lres.map(_._2))
      } yield algorithms.Joins.joinSet(lres, rres)

      case USDistinct(r) => for {
        rres <- recurse(r, left)
      } yield rres.filter{case (a, b) => a != b}

      case USId(_) =>
        left.map(x => (x, x)).right

      case USRel(rel) =>
        left.map(_.getRelatedMemoryObjects(rel, tree)).flattenE

      case USRevRel(rel) =>
        left.map(_.getRevRelatedMemoryObjects(rel, tree)).flattenE

      case USUpto(n, rel) =>
        val stepFunction =
          left => findPairsSetImpl(rel, Set(left), tree).map(_.mapProj2)
        upTo(stepFunction, left, n)

      case USFixedPoint(rel) =>
        // find a fixed point
        val stepFunction =
          left => findPairsSetImpl(rel, left, tree).map(_.mapProj2)
        for {
          res <- fixedPoint(stepFunction, left)
        } yield res

      case USExactly(n, rel) =>
        val stepFunction =
          left => findPairsSetImpl(rel, Set(left), tree).map(_.mapProj2)
        FixedPointTraversal.exactly(stepFunction, left, n)

    }
  }
```

E.2 FindSingle

```
def findSingleSetImpl(
    t: UnsafeFindSingle,
    tree: MemoryTree
): MemoryEither[Set[MemoryObject]] = {

    def recurse(t: UnsafeFindSingle) = findSingleSetImpl(t, tree)


    t match {
      case USFind(pattern) =>
        tree
          .getOrError(
              pattern.tableName,
              MemoryMissingTableName(pattern.tableName)
          ).flatMap(_.find(pattern).map(_.toSet))
      case USFrom(start, rel) => for {
        left <- recurse(start)
        res <- findPairsSetImpl(rel, left, tree).map(_.mapProj2)
```

```
      } yield res
      case USAndS(left, right) => for {
        r1 <- recurse(left)
        r2 <- recurse(right)
      } yield r1 intersect r2

      case USOrS(left, right) => for {
        r1 <- recurse(left)
        r2 <- recurse(right)
      } yield r1 union r2
    }
  }
```