Dhruv Makwana

Exploring the structure of mathematical theories using graph databases

Computer Science Tripos, Part II

Trinity College

March 3, 2017

Proforma

Name:	**Dhruv Makwana**
College:	**Trinity College**
Project Title:	**Exploring the structure of mathematical theories using graph databases**
Examination:	**Computer Science Tripos, Part II, 2016–2017**
Word Count:	–
Project Originator:	Dr. Timothy G. Griffin
Supervisor:	Dr. Timothy G. Griffin

Original Aims of the Project

100 words

Work Completed

100 words

Special Difficulties

100 words [hopefully "None."]

Declaration

I, Dhruv Makwana of Trinity 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:

Date:

Contents

1 Introduction

This dissertation offers a solution to problems regarding the presentation of mathematics. Firstly, these problems and the specific systems involved are described. Then, the aims of the project are stated; in later chapters, details of the prepration and implementation carried out are expounded. Lastly, the chapters on evaluation and conclusion provide evidence for the success of the project, reflections on the process and suggestions for future work.

1.1 Problem

Mathematics textbooks aimed at professionals/researchers follow a well-established rhythm: define some constructions and some properties on them and prove theorems on both, with lemmas, corollaries and notation interspersed throughout. Such a presentation is concise but limiting: it is linear; it forces the reader to keep track of dependencies such as implicit assumptions, previously defined results and the types and conventions behind any notation used; and it offers little opportunity to consider and compare different approaches for arriving at a result (i.e. number of assumptions, number of steps, some notion of the importance of a result such as number of uses by later results).

With the increasing popularity of interactive proof-assistants such as Coq [6] and Isabelle [8], many mathematical theories (such as the formidably large Feit-Thompson Odd Order Theorem [9, 1]) have been [3] or are being translated and formalised into machine-checked proof-scripts. However, these proof-scripts on their own inherit the same disadvantages as the aforementioned textbooks, as well as some new ones: they are usually more verbose and explicit and are primarily designed for automation/computation than readability. The former (usually out of necessity to convey to the computer the intended meaning) leads to unnecessary "noise" in the proof and the latter departs from the vocabulary or flow of a natural-language presentation.

The database world is currently experiencing a tremendous explosion of creativity with the emergence of new data models and new ways of representing and querying large data sets. *Graph databases* have been developed to deal with highly connected data sets and path-oriented queries. That is, graph databases are optimised for computing transitive-closure and related queries, which pose a huge challenge for traditional, relational databases.

1.2 Solution

A graph-based approach to the representation and exploration of the structure of proof-objects would be a far more natural expression of the complex relationships (i.e. chains of dependencies) involved in constructing mathematical theories. Questions such as "What depends on this lemma and how many such things are there?" or "What are the components of this definition?" could thus be expressed concisely (questions which are not even expressible with standard relational databases systems such as SQL). A popular graph database, Neo4j [7] with an expressive query language *Cypher* will be used for this project.

1.3 Coq Proof-Assistant

The Coq proof-assistant – implemented in OCaml – can be viewed as both a system of logic – in which case it is a realisation of the *Calculus of Inductive Constructions* – and as a *dependently-typed* programming language. Its power and development are therefore most-suited and often geared towards *large scale* developments.

On the logical side, Coq lays claim to projects such as the Four-Colour Theorem [2] (60,000 lines) and the aformentioned *Feit-Thompson* theorem (approximately $170,000$ lines, $15,000$ definitions and $4,200$ theorems) are feats of modern software-engineering.

On the programming language side, Coq has served as the basis for many equally fantastic projects. The *CompCert Verified C Compiler* [5] demonstrates the practical applications of theorem-proving and dependently-typed programming by implementing and proving correct an optimising compiler for the C programming language. *DeepSpec* [10], a recently announced meta-project, aims to integrate several large projects such as *CertiKOS* (operating system kernels, *Kami* (hardware), *Vellvm* (verifying LLVM) and many more in the hopes to provide complete, *end-to-end* verification of real-world systems.

1.4 Neo4j Database and the Cypher Language

Neo4j is a graph database system implemented in Java. Traditional, relational database theory and systems are designed with the goal of storing and manipulating information in the form of *tables*. As such, working with highly interconnected data, such as social network graphs is best tackled with the alternative approach of *graph databases*.

Briefly, a (directed) *graph* is defined as $G = (V, E)$ where V is a set of vertices or *nodes* and $E \subseteq V \times V$ is a set of edges or *relationships* between two nodes. A *graph database* is an OLTP (online transaction processing, meaning operated upon live, as data is processed) database management system with CRUD (create, read, update and delete) operations acting on a graph data model. Relationships are therefore promoted to first-class citizens and can be manipulated and analysed.

1.4.1 Cypher: An Illustrated Example

Cypher features heavy use of pattern-matching in an ASCII-art inspired syntax. The following (slightly contrived but hopefully illuminating) example in Listing 1 illustrates some of the key strengths of graph-based modelling using Cypher.

Suppose we have a puppy named "Cliff" looking for the nearest and most familiar children (for this example, a person under the age of six) to play with.

To see how Cliff (indirectly) likes/knows this child, we bind *path* to the result of

```
MATCH path = shortestPath(
  (puppy:dog {name: "Cliff"})-[:LIKES|:KNOWS*..4]->(child:person))
WHERE puppy.age <= 2 AND child.age < 6
RETURN path,
       child.name AS name,
       ORDER BY other.distance_from_clifford
```

Listing 1: Example Cypher Query

the *shortestPath* query. For the path itself, we start with a node following this structure: `(var:label {attrib: val})`. We then have a *labeled, transitive* relationship (explicitly limiting our search to paths of up to length four) expressed as an arrow with a label `-[..]->`. As such, we can discard any paths with relationships we do not want (e.g. `HATES`).

To filter based on more complex logic (than possible by pattern-matching directly on labels and attributes) we can express the requirement that the age of a dog by the name of Cliff be less than or equal to two (and similarly for the age of the child) in the `WHERE` clause.

Lastly, we return the path and order the results by proximity as a row of results, renaming the column of the child's name to simply "name".

1.5 Related Work

Some existing tools offer part of the solutions.

dpdgraph (github.com/Karmaki/coq-dpdgraph) is a tool which analyses dependencies between *compiled* Coq proofs. As such, desirable information about notation, tactics, definitions and the relationship between a type and its constructors is lost.

Coqdep is a utility included with Coq which analyses dependencies *at the module level* by tracking `Require` and `Import` statements.

Coq SerAPI (github.com/ejgallego/coq-serapi) is a work-in-progress library and communication protocol for Coq designed to make low-level interaction with Coq easier, especially for IDEs. It has a starting point for gathering some statistics of proof-objects in a project.

1.6 Aims of the Project

This project aimed to:

- represent Coq libraries as Neo4j graph databases, which involved

  - exploring and choosing the correct model
  - converting and extending existing code to output CSVs
  - writing new programs to extract extra information (omitted from other, existing tools)
  - writing new programs to automate database creation; and to

- create a library of Neo4j queries, intended
  - to highlight the structure and relationship between proof-objects
  - by coalescing and implementing several graph-related metrics.

1.7 Summary

An explanation of the problems which conventional presentations of mathematics suffer from was given, with *graph databases* proposed as a solution. Existing tools were mentioned and the requirements for a successful project were listed.

2 | Preparation

Before commencing implementation of the project, careful consideration was given to planning it. Current solutions were explored, studied and evaluated against the aims described in Section 1.6. The rest of this chapter will explain the initial set-up, elaborate on related work and outline the project's starting point.

2.1 Project Planning

This project presents a unique idea and breaks new ground. As such, the methodology had to suit and reflect the largely exploratory nature of the process. A spiral software development model was chosen: think of an idea, modify the model (of Coq proof-objects), implement and propogate the necessary changes, evaluate the end-result and repeat. This allowed for experimentation of ideas and flexibility of implementation strategies.

Git (`git-scm.com`) and GitHub (`github.com/dc-mak`) were invaluable during the project, allowing for easy tracking, reverting, reviewing and collaborating. New features could be tested on new branches before being merged in and a copy of the work was safely backed up in one more place. GitHub extensions such as Travis-CI (continous integration) were added in later, as it became apparent that precisely specified versioning, build-dependencies and automated tests were useful in spotting errors early.

2.2 Requirements Analysis

Several components of this project needed to function correctly, both individially and in conjuction for it to work. Separate parts for modelling/translation (from Coq to the chosen model), displaying and interacting (Neo4j/Cypher) and computation (Neo4j/Cypher plugins) needed to be developed and brought together. Below is a list of required features used throughout development to guide and provide context for implementation decisions.

- **Modelling**: The model should

  M1 include as much relevant data as possible. Here, relevant means useful to understanding a large library, but not so much so as to obfuscate any information or make learning how to use the project more difficult.

  M2 be flexible to work with and easy to translate. One could imagine different front-ends for interacting with and computing data from the model.

  M3 strike a balance between size and precomputing too much data. Figuring out which pieces of data can be reconstructed later and which are beneficial to compute during modelling will be a matter of experimentation and weighing up ease of implementation versus ease of later processing.

- **Interaction**: Interacting with the model should

  I1 primarily, allow users to understand the data. The following two points follow from this principal goal.

I2 support both graphical and textual modes of use. Small queries and novice users are likely to benefit from the presence of a well-designed GUI. However, larger queries requiring more computation and flexibility will benefit from a traditional, shell-like interface.

I3 be interactive and extensible. A static presentation of data, even in a GUI, would fail to make full use of graph-databases and the ability to query, in whatever way the user desires, information dynamically.

- **Computation**: Working with the model's data should

C1 be enabled by a core library of good defaults. Certain, common functions should be ready 'out-of-the-box' and provide users all they need to get started.

C2 allow the user to add their own functions. It is not possible to imagine and implement all the functionality users may desire and so a way to extend the project to suit their own needs would be of great use.

2.3 Technologies Used

Choice of implementation languages was, although an important decision, almost completely dictated by the programs at the core of the project (Coq and Neo4j).

Coq and its plugins —specifically, dpdgraph, which was used as a starting point for extracting information about Coq proof-objects from compiled proof-scripts —are written in OCaml (`ocaml.org`). Since it almost always wiser to work with and modify existing sytems (and more representative of real-world work) and as a functional language, OCaml benefits from strong, static (and inferred) type-system (allowing for easy experimentation, greater confidence in correctness), sticking to it for other parts of the tools which need not necessarily be in OCaml (e.g. the `dpd2` utility) was a welcome and easy decision. OCaml has several other benefits too, such as inductively-defined datatypes (useful for manipulating Coq constructs) and good editing tools.

Similarly, Neo4j and its plugins are (usually) written in Java, but several lanaguages are supported for the latter, both by Neo4j officially and by the community. As will be explained in Subsection 2.4.3, Java and R were found to be the most suitable for achieving this project's goals.

2.4 Starting Point

2.4.1 Coq

Coq is a *large* project, devloped by INRIA (France), and its size and complexity are best experienced through detangling the source code for oneself. Just for

the implementation of the system (not including the standard library), Coq features approximately 3 major ASTs, 6 transformations between them, 3000 visible types, 9000 APIs and 521 implementation files containing 228,000 lines of dense, functional OCaml.

However, most of this massive project is sporadically (and tersely) documented. Even after consulting the Coq developers' mailing-list, several hours were spent browsing the source code to overcome the severe difficulties in understanding the project. Prior familiarity with *using* Coq (as an introduction to tactical theorem-proving and dependently-typed programming) was not useful for understanding the internals beyond context and how to compile and use programs and libraries. However, it did serve as invaluable insight for *cognitive walkthroughs* carried out during the implementation phase (as a structured way of guiding the design of the model and libraries).

2.4.2 Existing Tools for Coq

A number of tools were studied to learn their approaches and analyse their strengths and weakeness. A full, detailed comparison between all the tools mentioned and this project will be presented later, during evaluation. What follows is a brief overview of each tool and the reason it was insufficient for the purposes of meeting the project aims and requirements.

Coqdoc

Coqdoc is a documentation tool for Coq projects, includes as part of the Coq system.. It can output to raw text, HTML, LaTeX and a few other formats. Although it supports prettifying code with syntax highlighting and unicode characters, its most relevant feature was its hyperlinking: potentially useful for building dependency graphs.

However, the whole tool worked on an entirely *lexical* level, with no formal parsing, understanding or elaboration of the code structure. Some efforts were made to modify its output into a useful format (e.g. comma-separated values) for other tools; however these did not prove fruitful because tokenisation cannot infer or preserve as much information as full compilation. Hence, since it could not meet any of the modelling requirements (completeness M1, flexibility M2 and size/precomputation M3) this approach was abandoned.

Coqdep

Coqdep is a tool which computes inter-module dependencies for Coq and OCaml programs, outputttting in a format readable by the `make` program. Although on first impressions, this tool seemed to offer more flexibility than coqdoc, it was even more restrictive: it simply searches for keywords (such as `Require` or `Import` for Coq and `open` or dot-notation module usage for OCaml) per file and outputs them

accordingly. As with coqdoc (and for the same reasons), this approach was also abandoned.

CoqSerAPI

Coq Se(xp)rialized API is a new library and communication protocol aiming to make low-level interactions easier (using OCaml datatypes and s-expressions), particularly for tool/IDE developers. While this is likely to be useful in the future, it is still far from complete and is more geared towards interactive *construction* (via a tool/IDE) rather than *analysis*. As such, tracking dependencies (critical to the modelling requirements) is not possible.

dpdgraph

dpdgraph is a tool which extracts dependencies between Coq objects from compiled Coq object-files to a `.dpd` file. It includes two example tools: `dpd2dot` (for producing a `.dot` file for static visualisation) and `dpdusage` (for finding unused definitions). Its developers intended it to be a starting point for tools to build upon.

Although lots of information such as notation, the relationship between constructors and the types they construct, proof tactics, the precise kind of an object (e.g. fixpoint, class, lemma, theorem, etc.) and which module an object belongs to was missing, it seemed unlikely that the information was not present in the compiled object files. Assuming that the data was already present in those files, but simply *ignored or unused*, implementation of the modelling aspect of this project focused on understanding and augmenting dpdgraph to add the missing pieces to the model and convert the whole thing to comma-separated values (henceforth referred to as CSVs)

2.4.3 Neo4j

Neo4j is one of the most popular graph database systems. It supports both graphical and textual modes of use and is easily extensible (through Cypher plugins and several language-specific bindings and libraries). It meets all the interaction requirement of helping users to understand data, being flexible in its use and extensible in its capabilities. It even includes a tool to import CSVs files containing nodes and edges into a new database, which meant modelling could be focused towards extracting and expressing in a simple format as much information as possible.

Neo4j also includes an interactive graphical interface, accessible through an ordinary web-browser. As can be seen on Figure 2.1, the tool offers

- an overview of the current labels, relationships and properties in the database

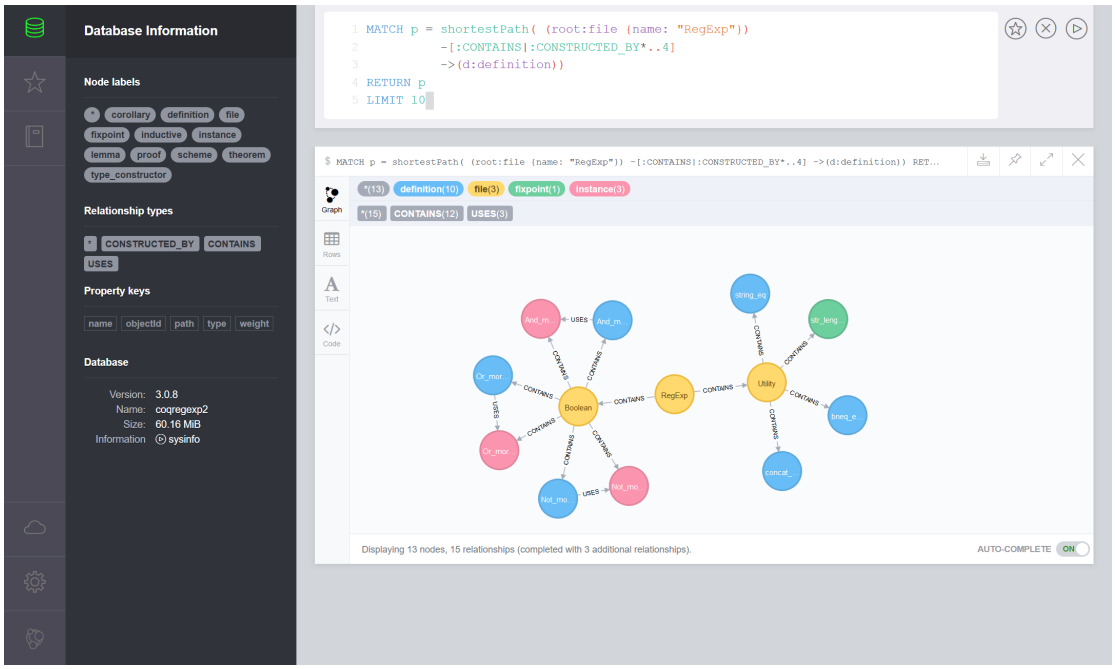


Figure 2.1: Neo4j Interactive Browser

- syntax-highlighted interactive-editing box
- graphical representation of query result (with options to view it as rows like a shell, or raw JSON text results) with profiling information along the bottom
- easy access to favourite queries and scripts (the star on the left)
- easy access to documentation and system information (the book on the left)
- and many more features such as browser sync, settings and the 'about' section.

2.4.4 Existing Tools for Neo4j

APOC: Awesome Procedures on Cypher

Awesome Procedures on Cypher, or *APOC* for short, is a community-maintained Java plugin featuring several graph algorithms callable from within Cypher itself. Although there are other extension libraries (such as MazeRunner), APOC is well documented, up-to-date and the most comprehensive, and therefore the obvious choice as a foundation. By being a Java library hooked into Cypher, it offered the potential for additional functionality to be built on top of it which packaged-up some of the more complex features into *domain-specific* queries, intended for Coq users not familiar with Neo4j to get started with. Thus, APOC helps step towards meeting the *interaction* requirements for this project by being easy to

understand, flexibile to use and extensible; even going part-way towards meeting the *computation* requirements.

igraph

APOC has some key strengths that made it a good choice: it is easy to install and use and has some basic graph algorithms to get started with. However, its main focus is on interacting with and combining different sorts and sources of data and so lacks graph analysis functionality *beyond* the basics. The fact that it is implemented in Java further adds to its limitations: it is not well-suited to more intense analyses over large graphs of libaries and is insufficient to *fully* meet the *computation* requirements of this project.

For such tasks, igraph is ideal: it is described on its website as a *collection of network analysis tools, with the emphasis on efficiency, portability and ease of use*. Written in C/C++ (with bindings for R and Python), igraph offers a *comprehensive* set of graph algorithms without sacrificing on performance. These alogrithms and their uses will be described later, in the Implementation chapter. For now, it suffices to surmise that although igraph is not as easy to interact with (via the statistics-oriented programming language R, as detailed in the next paragraph) as APOC, the extra capabilities afforded were indispensible towards achieving the *computation* requirements of a core library of good defaults.

visNetwork

With igraph and APOC providing starting points for the computational aspects of the projects, and the interactive Neo4j browser providing a well-polished, graphical mode of interaction with basic, but useful, visualisation, the last piece of the project was to incorporate the extra information gained from *executing* the graph algorithms.

Several visualisation programs exist for Neo4j; however, many are for commercial, industrial use and offer the features/complexity (and pricing) to match. All tools which offer live visualisation with built-in Cypher query execution (e.g. KeyLines, TomSawyer, Linkurious) are proprietary, requiring a fee to use and offering more granularity than required. Offline (and open-source) solutions (which require data to be exported in some manner before visualisation) such as Gephi or Alchemy.js offer similarly many features, but at the cost of a steep learning curve. Ultimately, visNetwork, (an R library exporting to JavaScript which can be rendered inside a browser) was chosen due to its simplicity and ease of integration with previous tools mentioned above.

R

R is a statistics-oriented programming language, part of the Free Software Foundation's GNU poject. It is relevant for this project because it offers an easy way to

tie together Neo4j (through official bindings), igraph and visualisation using visNetwork. This convenience came at the price of having to learn R for this project, having been unfamiliar with it prior. Nonetheless, it is a well-documented, relatively easy to pick-up language and offered even more opportunities for learning during the course of this project.

2.5 Summary

A detailed account into the planning of this project was given. The choice of development methodology (spiral) and development tools (Git, GitHub, Travis-CI) were noted. Requirements on modelling, interaction and computation were explained and the choice of technologies and tools used as statrting points (Coq, OCaml, dpdgraph, Neo4j, APOC, igraph, R and visNetwork) were justified *in relation* to *which* requirements they satisfied.

3 Implementation

Upon completion of the project's plan, its execution commenced. What follows is an account of the programs written, problems encountered, solutions implemented and tests conducted using the project-structure shown in Figure 3.1 as a guide. *Reasons* for design decisions (arrived at using *formative* evaluation techniques) are detailed in Chapter 4 on Evaluation.

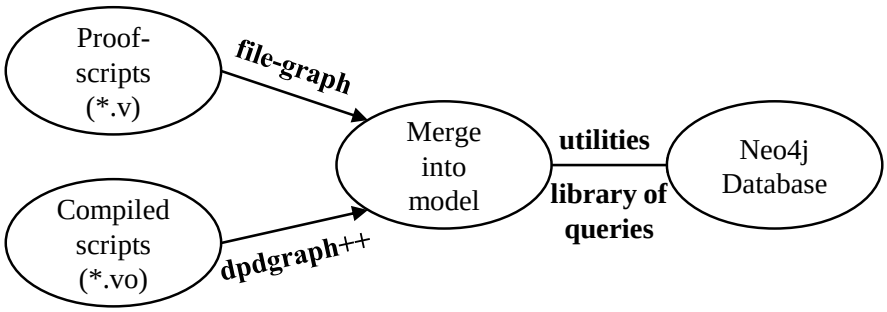


Figure 3.1: System Components

3.1 Coq object-files to CSV

This section of implementation corresponds to "dpdgraph++" on Figure 3.1: modelling the data contained in and the structure of Coq object-files (`*.vo`) as comma-separated values (CSVs). The initial model (inherited from the open-source "dpdgraph" tool) and subsequent changes to it will be described.

3.1.1 Modelling

Initially, each edge was assigned a *weight* representing the number of (directed) uses of one node by another. Each node was assigned four *properties*:

- *body*, a boolean representing whether a global declaration was either transparent or opaque;
- *kind*, a ternary value representing whether a *global reference* (a kernel side type for all references) was a reference to either the environment, an inductive type or a constructor of an inductive type;
- *prop*, a boolean value representing whether a term is a `Prop` (a decidable, logical property about a program, as opposed to a general `Type`);
- *path*, a string value represent the module an object is in.

These properties were difficult to understand: they are not in the vocabulary of a Coq programmer (e.g. `Definition`, `Inductive`, `Theorem`) and could not represent the richness of the AST appropriately. It was not documented how to translate these constructs back to familiar terms and thus, it quickly became clear that the these properties needed to be replaced by more general and descriptive ones.

Precise Kinds

Apart from *path*, all the properties were removed and replaced by two *labels*: labels are used to group nodes into subsets; since a node can belong to more than one subset, it can have more than one label assigned to it. Implementation of this was straightforward: simply a matter of looking up and expanding the abstract syntax tree starting from the type `global_reference`.

The two labels are *kind* and *subkind*. A *kind* is a string which can take one of the following values, each directly corresponding to an AST term: module, class, type_constructor, inductive_type, definition, assumption or proof.

Optionally, some terms have *subkinds*, for distinguishing different constructs more precisely. For example, when writing Coq, there is no `Proof` keyword; instead `Theorem`, `Lemma`, `Fact`, `Remark`, `Property`, `Proposition` and `Corollary` all are *synonyms* for proofs. Full details can be found in Appendix A.

Recursive Modules

What remained from the initial model was the *path* property. The issue here was that an inherently *hierarchical* structure (of inclusion) was represented *flatly* as a string attribute of a node. This made modules second-class citizens, not subject to the same analyses and manipulations as proof-objects, excluding the possibility of expressing *module-level dependencies* (as possible in the coqdep tool).

Implementing this feature was difficult; there were two major phases. Firstly, the type of a node was exanded to include modules (using a variant datatype). Finishing this was just a matter of locating and fixing the resulting type-errors. This meant modules were in the model, but as a flat structure: modules could be related to objects but not to other modules.

Thus, the second major phase was inferring and adding all the "ancestors" of a module with the correct relationships (parent as source, child as destination, repeatedly up to the root module). The initial attempt resulted in stack-overflow for larger examples. A stack-trace did not highlight any point of error so it was *assumed* to be due to the naïve, but easy-to-write (and check) non-tail-recursive implementation. So, the code was rewritten in a space-efficient tail-recursive manner (which allows for deeply nested module hierarchies to be handled). Surprisingly, the problem persisted; further investigation discovered the fault lay in the separate `dpd2` tool (the details of which can be found in Subsection 3.1.2).

Inductive Types and Constructors

Now that the properties of the original dpdpgraph had been superseded by more general and flexible alternatives, it was time to implement new features. One of the most obvious and frustrating omissions from the initial model was the inability to relate an inductive type to to its constuctor(s). Expanding the AST term for type-constructors showed which type it constructed. However,

Types

Conducting a *cognitive walkthrough* (detailed in Chapter 4) spurred the addition of *type signatures* to the model. Type theory is central to a Coq user's work and being able to include them in the model, would, along with kinds, subkinds and modules, help towards meeting the modelling requirement M1 of including as much relevant data as possible.

Getting the type was unexpectedly straightfoward; following the functions called when the Coq command `Check` <*expression*> (for printing the type of a given expression) lead to the algorithm; all that was left was converting the output to an OCaml string. It was *using* the output which was problematic. Newlines, quotation marks, and commas had to be replaced by hash signs, single-quote marks and underscores respectively, so as to not interfere with the `.dpd` and CSV encoding of data.

Relationships

Relationships were the most interesti Intention, execution, problems (remember the stack overflow), solutions.

- Tried and *removed* x_USED_BY_y
- USES
- CONTAINS
- CONSTRUCTED_BY
- Emphasis on direction of arrow for exploration, importance of consistency

3.1.2 Translation

3.2 Coq source-files to CSV

Intention, execution, problems, solutions.

Also, how it fit in with larger, open-source projects.

3.2.1 Missing Information

3.2.2 Collection

Explain Glob Files.

3.2.3 Merging

3.3 CSV to Neo4j

Intention, execution, problems, solutions.

Also, how it fit in with larger, open-source projects.

3.3.1 Neo4j Import Tools

3.3.2 Impact of Changes to Model

Trade-offs, execution.

3.4 Query Library

Intention, execution, problems, solutions.

Also, how it fit in with larger, open-source projects.

3.4.1 APOC

3.4.2 Additions on top of APOC

3.4.3 igraph

Terminology

Betweenness Centrality

Closeness Centrality

PageRank

Community Detection

3.4.4 Visualisation

3.4.5 R Library

3.5 Project Related

Modifying make files, testing, continuous-integration builds, editors

3.6 Summary

Features implemented, dead-ends and lessons learnt.

4 | Evaluation

Table 4.1: Comparison of Features

	Property 1	Property 2	Property 3
System 1			X
System 2	X	X	X
System 3	X		X

4.1 Constructs Translated

4.1.1 Cognitive Walkthrough

4.1.2 Possible Extensions

4.2 Library of Queries

4.3 Sample output

4.3.1 Small: CoqRegExp

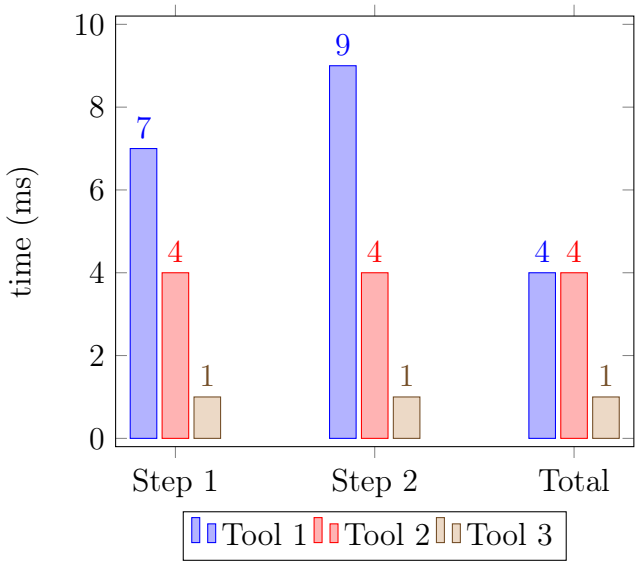
4.3.2 Large: MathComp

4.4 A Comparative Analysis

4.4.1 Features

4.4.2 Performance

Figure 4.1: Comparison of Execution Times



5 Conclusions

5.1 Summary

5.2 In Hindsight

5.3 Future work

Bibliography

[1] Helmut Bender, George Glauberman, and Walter Carlip. *Local analysis for the odd order theorem*, volume 188. Cambridge University Press, 1994.

[2] Georges Gonthier. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[3] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, et al. A machine-checked proof of the odd order theorem. In *International Conference on Interactive Theorem Proving*, pages 163–179. Springer, 2013.

[4] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, Inria Saclay Ile de France, 2015.

[5] Xavier Leroy. The compcert c verified compiler. *Documentation and user's manual. INRIA Paris-Rocquencourt*, 2012.

[6] The Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2004. Version 8.0.

[7] Neo4j. Neo4j. neo4j.com. Accessed: 13/10/2016.

[8] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic.* Springer-Verlag, Berlin, Heidelberg, 2002.

[9] Thomas Peterfalvi. *Character theory for the odd order theorem*, volume 272. Cambridge University Press, 2000.

[10] Benjamin C Pierce. The science of deep specification (keynote). In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, pages 1–1. ACM, 2016.

A | Full Model

B | Project proposal

Computer Science Tripos – Part II – Project Proposal

Exploring the structure of mathematical theories using graph databases

Dhruv C. Makwana, Trinity College

Originator: Dr. Timothy G. Griffin

Project Supervisor: Dr. Timothy G. Griffin

Directors of Studies: Dr. Frank Stajano & Dr. Sean B. Holden

Project Overseers: Dr. David J. Greaves & Prof. John Daugman

Introduction and Description of the Work

This project aims to (a) represent Coq libraries as Neo4j (graph) databases and (b) create a library of Neo4j queries with the goal of highlighting the structure and relationship between the representations of the proof-objects.

Mathematics textbooks aimed at professionals/researchers follow a well-established rhythm: define some constructions and some properties on them and prove theorems on both, with lemmas, corollaries and notation interspersed throughout. Such a presentation is concise but limiting: it is linear; it forces the reader to keep track of dependencies such as implicit assumptions, previously defined results and the types and conventions behind any notation used; and it offers little opportunity to consider and compare different approaches for arriving at a result (i.e. number of assumptions, number of steps, some notion of the importance of a result such as number of uses by later results).

With the increasing popularity of interactive theorem-provers such as Coq [6] and Isabelle [8], many mathematical theories (such as the formidably large Feit-Thompson Odd Order Theorem [9, 1]) have been [3] or are being translated and formalised into machine-checked proof-scripts. However, these proof-scripts on

their own inherit the same disadvantages as the aforementioned textbooks, as well as some new ones: they are usually more verbose and explicit and are primarily designed for automation/computation than readability. The former (usually out of necessity to convey to the computer the intended meaning) leads to unnecessary "noise" in the proof and the latter departs from the vocabulary or flow a natural-language presentation may have.

The database world is currently experiencing a tremendous explosion of creativity with the emergence of new data models and new ways of representing and querying large data sets. *Graph databases* have been developed to deal with highly connected data sets and path-oriented queries. That is, graph databases are optimised for computing transitive-closure and related queries, which pose a huge challenge for traditional, relational databases.

A graph-based approach to the representation and exploration of the structure of proof-objects would be a far more natural expression of the complex relationships (i.e. chains of dependencies) involved in constructing mathematical theories. Questions such as "What depends on this lemma and how many such things are there?" or "What are the components of this definition?" could thus be expressed concisely (questions which are not even expressible with standard relational databases systems such as SQL). A popular graph database, Neo4j [7] with an expressive query language *Cypher* will be used for this project.

Resources Required

Software

Several components of software will be required for executing this project, all of which are available for free online.

For using the proof-scripts, the Coq proof assistant will be required, as well as the Proof General proof assistant (`proofgeneral.github.io/`) for the Emacs (`www.gnu.org/software/emacs/`) text-editor.

For writing the plug-in to access Coq proof-objects, the parser and associated modules in the source code will be required (`github.com/coq/coq`) written in the OCaml programming language (`ocaml.org`) with the OCaml's Package Manager OPAM (`opam.ocaml.org`).

For building the library of (Cypher) queries, Neo4j Community Edition will be used.

Hardware

Implementation and testing will be done on both Windows 10 and a Linux Virtual Machine as appropriate and convenient on a Surface Pro 3 (Intel Haswell i7-4650U

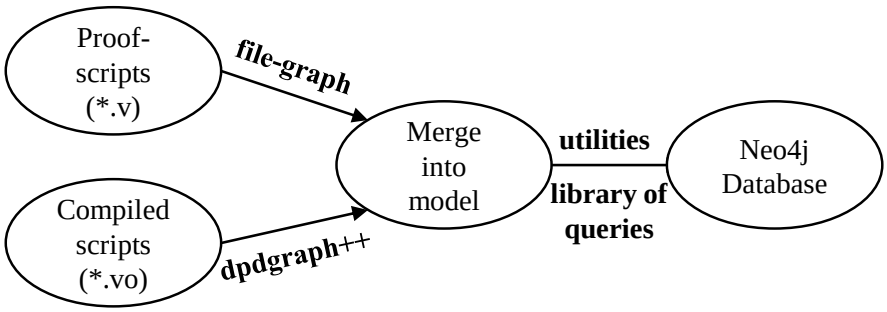


Figure B.1: System Components (repeated from page 16)

1.7-3GHz, 8GB RAM, 512GB SSD) with a personal GitHub account and physical backup drive (Seagate 1TB) making hourly backups using Windows' File History.

Starting Point

Some existing tools offer part of the solutions: these will be used and combined as appropriate. A large part of the project will rely on my knowledge of OCaml and Coq usage and internals.

Coq-dpdgraph (`github.com/Karmaki/coq-dpdgraph`) is a tool which analyses dependencies between *compiled* Coq proofs. As such, desirable information about notation, tactics, definitions and the relationship between a type and its constructors is lost.

Coqdep is a utility included with Coq which analyses dependencies *at the module level* by tracking `Require` and `Import` statements.

Coq SerAPI (`github.com/ejgallego/coq-serapis`) is a work-in-progress library and communication protocol for Coq designed to make low-level interaction with Coq easier, especially for IDEs. It has a starting point for gathering some statistics of proof-objects in a project.

All of these tools have the same disadvantage: they present information statically, with no way to query and interact with the information available.

Substance and Structure of the Project

The project will have three major parts, as shown in Figure 3.1.

Processing Compiled Files

First, using coq-dpdgraph as a starting point, a tool which expresses a compiled proof-script as CSV files (shown as "dpdgraph++" in the diagram). Finding what information can and should be extracted will be an iterative process. Although coq-dpdgraph is functional, it is very basic with no way of even relating the relationship between a (co-)inductive type and its constructors, hence much work is to be done to even come close to utilising the full potential of compiled proof-scripts.

Processing Source Code Directly

Second, using Coq's sophisticated extensible-parser, to parse, gather and convert to CSV files the desirable but missing information coq-dpdgraph does not extract (shown as "file-graph" in the diagram). An interesting feature of Coq's parser is that it allows new constructs and notation to be defined: this is used heavily in some projects and therefore poses a great challenge for simply understanding and using the parser effectively.

Extraction and Analysis Tools

Lastly, writing utilities to automate analysis of Coq files and importing them into Neo4j and libraries of queries to run on imported data in Neo4j. Since it is not known what sort of data can be extracted and what will be useful or interesting to know, modelling the data – in this case the structure and objects of a mathematical proof – will be an non-trivial task which will be tackled iteratively.

Extensions

Extensions for this project will come from the process of adapting the project to be compatible with SSReflect [4], part of the Mathematical Components set of tools for Coq. These set of tools use low-level hooks in the Coq plugin system to significantly alter the specification and computation of proofs. As such, although they allow for large-scale projects to be formalised more easily, they are non-standard and would thus be very difficult to support fully.

Success Criteria

Alongside a planned and written dissertation describing the work done, the following criteria will be used to evaluate the success of this project:

1. A schema of attributes and relations for each proof-object is defined.

2. Programs which convert proof-scripts and compiled proofs to CSV files are implemented.

3. A library of queries in order to manipulate and explore the proof-objects is implemented.

4. These new sets of tools are shown to have more capabilities and perform comparably to existing tools for exploring mathematical theories.

Timetable and milestones

Date	Milestone
21-10-2016	Complete Project Proposal
04-11-2016	Finish a prototype compiled-to-CSV tool. Get familiar with Neo4j Cypher. Understand how to use the Coq parser.
18-11-2016	Refine compiled-to-CSV tool: tests and documentation. Explore queries possible and start the library. Begin work on translating Coq constructs from proof-scripts.
02-12-2016	Finish a prototype script-to-CSV tool.
16-12-2016	Test and document script-to-CSV tool.
30-12-2016	Begin work on integrating tools into one workflow.
13-01-2017	Stabilise and document whole project so far. Prepare presentation for CoqPL Conference.
27-01-2017	Look at SSReflect and evaluate changes to be made.
10-02-2017	Incorporate changes from feedback/new features.
24-02-2017	Test and document the new features.
10-03-2017	Write Introduction, Preparation and Implementation chapters.
24-03-2017	Fix bugs/unexpected problems.
07-04-2017	Write Evaluation and Conclusion chapters.
21-04-2017	Fix bugs/unexpected problems.
05-05-2017	Complete Dissertation (references, bibliography, appendix, formatting).